BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2020 Bc. Jozef Urbanovský

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS
ÚSTAV TELEKOMUNIKACÍ

TRANSMISSION NETWORK EMULATOR
EMULÁTOR PŘENOSOVÉ SÍTĚ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Jozef Urbanovský**
AUTOR PRÁCE

SUPERVISOR — **Ing. Ondřej Krajsa, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study field **Communications and Informatics**

Department of Telecommunications

Student: Bc. Jozef Urbanovský ***ID:*** 187393

Year of study: 2 ***Academic year:*** 2019/20

TITLE OF THESIS:

Transmission network emulator

INSTRUCTION:

Design and implement an application for the Linux operating system that will serve as an emulator of the transport network over the Ethernet protocol. The application will transmit frames between two interfaces and will introduce delay, jitter, loss and modification of frames and data units on higher layers into the transmission. The application will be configurable.

RECOMMENDED LITERATURE:

[1] ROSEN, Rami. Linux kernel networking: implementation and theory. New York, NY: Apress, 2014, xxxii, 612 stran : tabulky, grafy ; 24 cm. ISBN 9781430261964.

[2] YANG, Lixiang. The Art of Linux Kernel Design. 1. Auerbach Publications, 2014. ISBN 9781466518032.

Date of project specification: 3.2.2020 ***Deadline for submission:*** 1.6.2020

Supervisor: Ing. Ondřej Krajsa, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
Subject Council chairman

ABSTRACT

Master's thesis deals with an implementation of a WAN network emulator operating over TCP/IP network stack. Work describes Linux network kernel stack, as well as, program realization of emulator with the use of userspace utilities, such as tc, netem, ifb and iptables, which are manipulating the kernel. The application for emulation is written in Python and it is controlled with graphical user interface made with PyQt5. The application was tested and verified to be accurate representation of specified WAN and to be resource efficient.

KEYWORDS

Linux, kernel, emulator, traffic control, quality of service, tc, netem, tbf, WEnux

ABSTRAKT

Diplomová práca sa zaoberá vývojom emulátoru prenosových parametrov dátovych sietí nad protokolovou sadou TCP/IP. Práca opisuje špecifiká sieťovej časti Linuxového jadra ako aj realizáciu emulátoru pomocou userspace programov ako tc, netem, ifb a iptables ovládajúcich jadro. Program pre emuláciu je realizovaný v jazyku Python a je ovládaný cez grafické rozhranie vytvorené s PyQt5. Aplikácia bola otestovaná a verifikovaná, že je dostatočne presnou reprezentáciou špecifikovanej WAN siete a že taktiež využíva prostriedky efektívne.

KĽÚČOVÉ SLOVÁ

Linux, jadro, emulátor, riadenie prevádzky, kvalita služieb, tc, netem, tbf, WEnux

URBANOVSKÝ, Jozef. *Transmission network emulator*. Brno, 2020, 65 p. Master's Thesis. Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Department of Telecommunications. Advised by Ing. Ondřej Krajsa, Ph.D.

ROZŠÍRENÝ ABSTRAKT

Experimenty sú dôležitou časťou vývoja každého sieťového protokolu. Ich hlavným použitím je validácia teoretických výsledkov, prípadne validácia praktických riešení v stave implementácie, ktorých testovanie nie je jednoduché namodelovať alebo naimplementovať. Výsledkom experimentov by mal byť posudok protokolu s ohľadom na jeho spoľahlivosť, robustnosť a funkcionalitu špecifikovanú definíciou. Ale samozrejme v niektorých prípadoch, ako napríklad v rozľahlej sieti typu WAN, to nie je možné, prípadne je veľmi obtiažne ju vytvoriť, nakonfigurovať a spravovať reálne testovacie prostredie s vysokou komplexitou.

V takomto prípade je možné použiť simulátor, ktorý používa funkcionálne modely pre všetky časti siete, ako aj samotnej aplikácie. Simulácia v porovnaní s reálnou sieťou môže byť len vo forme softwaru. Kvôli tomu je nutné, aby program modeloval správanie siete tak, že vypočíta všetky možné interakcie medzi rôznymi sieťovými entitami. Jedná sa o čisto matematickú reprezentáciu sieťovej prevádzky, protokolov a iných sieťových prvkov. Simulátory sú zvyčajne veľmi komplikované programy s rôznou úrovňou abstrakcie a podľa typu simulačného modelu majú aj obmedzenia, ktoré môžu byť ťažko prekonateľné.

Na druhej strane emulátory sa snažia spojiť simuláciu a experimentovanie v reálnom svete tak, že poskytnú kontrolované prostredie, v ktorom môže bežať skutočný kód. V porovnaní so simulátormi, emulátory môžu existovať vo forme softwaru aj hardwaru. Ich umiestenie v sieti je zvyčajne medzi dvoma segmentami v lokálnej sieti typu LAN, kde môžu replikovať spojenie a rozhľahlú sieť typu WAN bez nutnosti existencie fyzických zariadení či sieťovej prevádzky.

Cieľom tejto diplomovej práce je navrhnúť a programovo realizovať sieťový emulátor typu WAN pre systém Linux, a preto je nutné poznať ako funguje sieťový zásobník v tomto systéme. Linux využíva ISO/OSI referenčný model pre prevažnú väčšinu svojho sieťového zásobníka, ktorý je rozdelený do mnohých modulov jadra. Linuxové jadro, známe aj ako *kernel*, obsluhuje druhú až štvrtú vrstvu referenčného modelu. Po prijatí paketu na sieťovú kartu a jeho následným odovzdaním do sieťového ovládača, preberá kontrolu kernel. Hlavnou úlohou sieťového ovládača v kerneli je príjmať pakety, ktoré sú cielené pre daného účastníka a ich predávanie do vyšších vrstviev, prípadne odosielanie paketov generovaných lokálne, cielených do iných sietí a taktiež preposielanie sieťovej prevádzky, ktorá nie je určená danému účastníkovi. V tretej vrstve referenčného modelu sa nachádza najväčší komponent sieťovej časti kernelu vo forme *netfilter* subsystému. Netfilter je sieťová štruktúra na zachytávanie volaní generovaných v rôznych častiach sieťového zásobníka a ich prípadnú obsluhu. Časťou tohto susbystému je aj *iproute* modul, ktorý má na starosti ri-

adenie tretej vrstvy referenčného modelu a taktiež obsahuje submodul na ovládanie prevádzky. Modul *tc* je najdôležitejšou časťou diplomovej práce, nakoľko sa oň opiera implementácia samotnej aplikácie emulátoru. Jedná sa o takmer poslednú časť sieťového zásobníka pred tým, než sú pakety zaslané sieťovému ovládaču, prípadne jedným z prvých segmentov pre prichádzajúce pakety z vyšších vrstiev. Jeho hlavnou úlohou je rozhodovanie o prechádzajúcich paketoch, či ich prijať alebo zahodiť, prípadne akou rýchlosťou, akým poradím, atď. Pre účely diplomovej práce sú vytvorené pravidlá pre modul tc, ktoré sú určené na degradáciu siete v špecifických parametroch, aby bolo možné emulovať sieť WAN. Hlavné časti ovládania prevádzky sú klasifikátor a plánovač.

Vďaka natívnemu Linuxovému prostrediu s plným prístupom do systému je možné využiť modul tc. Pre overenie konceptu, že bude možné vytvoriť spomínaný emulátor za pomoci tohto modulu, bolo nutné vykonať experimenty s využitím časti modulu tc - *netem*, ktorý bol schopný ovplyvnovať parametre, ako napríklad latenciu, strátovosť, jitter a mnohé iné. Nakoľko sa pre samotnú manipuláciu bude používať priamo kernel, výber programovacieho jazyka z hľadiska rýchlosti nie je nutné riešiť a pre implementáciu bol zvolený jazyk Python. Prvotná implementácia pozostávala z využitia plánovača z modulu tc na odchádzajúce pakety zo systému. V prichádzajúcom smere nebolo možné využiť plánovač, nakoľko pakety v tomto smere neprechádzajú danou časťou netfilter subsystému, ktorý by ich mohol zachytiť. Pre prípad prichádzajúcich paketov bol využitý virtuálny IFB blok, ktorý slúžil na zachytávanie a presmerovávanie prichádzajúcich paketov, kde už bolo možné aplikovať plánovač.

Finálna aplikácia dovoľuje užívateľovi vytvoriť jednu sieť za pomoci sieťového mosta z dvoch daných sieťových rozhraní alebo prípadne použiť už existujúce rozhranie pre emuláciu. Pre ovládanie samotných parametrov programu, užívateľ využije grafické rozhranie nadizajnované pomocou *Qt Designeru*, ktoré je kontrolované s knižnicou *PyQt5*. Aplikácia má minimálne závislosti na externé knižnice a programy.

Testovanie a verifikácia sú poslednou časťou diplomovej práce. Sú zamerané na presnosť implementácie z pohľadu parametrov a využívania zdrojov, hlavne procesorového času a alokovanej operačnej pamäti. Presnosť je v rámci softwarových možností vysoká a jedinou prekážkou k dokonalej situácii je granularita času v kerneli. Použité prostriedky sú efektívne využité a nedochádza k žiadnemu plytvaniu zdrojov. Je nutné poznamenať, že pri veľmi nízkych hodnotách parametrov, je presnosť simulácie nižšia, ale pri použití parametrov blízkych reálnej WAN sieti je možné hovoriť o zanedbateľných rozdieloch.

DECLARATION

I declare that I have written the Master's Thesis titled "Transmission network emulator" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

I would first like to thank my supervisor Ing. Ondřej Krajsa Ph.D. of the Faculty of Electrical Engineering and Telecommunications, Brno University of Technology, for providing valuable knowledge, guidance and patience.

I would also like to acknowledge my co-worker Ing. Ondrej Licthner for his insight and help along the way, even though he had to read half-baked ideas and correct it numerous times.

Finally, I must express my very profound gratitude to my friends and family, especially Andrea and Adrián for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

Listings

Introduction

Experimenting is an essential part of the development of any network protocol. Validation of theoretical results and their extension to practical solution, which might not be easy to model or implement, can be a main use of experiments. Primary outcome of a sequence of tests should be an assessment of the protocol in terms of reliability, robustness and proper functionality, described by its definition. However, in many cases, such as wide-area network, it might be impossible to build, maintain and configure real world tests or testbeds[1], due to their nature, which can be extremely complex. Core components of the experimental environment are following: the application using the protocol for evaluation and the network on which the protocol is run.

As a valid option, one can use a simulator, which employs functional models for all parts of the network, as well as, the application itself. Simulation, compared to a real network, runs only as a software. Due to this fact, the program has to model the behaviour of a network by calculating the interaction between the different network entities. It uses purely mathematical representation of traffic, channels and protocols. Nonetheless, main restraints of the software simulation are in the complexity of the implementation, nature of the simulation, which can be either continuous or discrete, level of abstraction, validation process of the software itself. All of these factors create a simulator with predefined level of performance, realism and simplicity.

On the other hand, emulators are trying to bridge the gap between simulation and real system experiments by providing controlled environment on which real code can run. Opposed to simulators, they can both exist in the form of hardware and software solutions. Network emulators are usually physically placed between two LAN segments, where they can replicate a client/server WAN connection, without the need for a physical devices to be present, or even live traffic. Typically, emulators are configured to manipulate bandwidth constraints, apply impairments, being packet loss, delay, jitter and many others. With wide-area network-like configuration application performance and end-user experience can then be observed, tested and validated under conditions, which are emulating the real world as closely as possible [1].

This thesis analyzes Linux kernel network stack, with most of its relations and interactions between individual layers. Its primary concerns are data link layer, network layer and transport layer, which are handled by the kernel. Network device, that represent data link layer, is being discussed as well, with its methods of dealing with egress or ingress packets and its interactions with network layer. Going upwards

[1]Platform for conducting rigorous, transparent, and replicable testing of scientific theories

in network stack hierarchy, thesis debates *netfilter* modules as a whole, along with its specific sub-modules. Main focus is directed to traffic control, explaining its role in networking, as well as, concepts which are used in the WEnux implementation. All sections discussing Linux kernel also contain in-depth explanations through code examples from kernel itself or graphs explaining their relations and functionality.

Later chapter 2 explains emulator's relation to other types of network virtualization and motivation for choosing the emulation, instead of simulation. Importance of this chapter is critical, as it allows one, to understand the limitations that emulation has and its difference between simulation and real scenario.

Chapter 3 is meant for the WEnux emulator itself, with its research of existing solution, such as WANem, which served as an inspiration, used technologies previously explained in the chapter 1. Final application is discussed, with its graphical user interface, its functionality, implementation and lastly UML model of the Python utility.

Last chapter is dedicated to tests and evaluation of WEnux emulator, with regards to its accuracy of emulation and resource usage. Basic testing methodology is established and according to it, the results are presented.

1 Linux network stack

First chapter provides comprehensive knowledge of Linux networking for understanding the main concepts of the thesis. It mainly describes *netfilter* with almost all of its components and packet traversal through the network stack, explaining step by step actions the packet takes. There is also emphasis on a routing subsystem, as well as, traffic control in both general and Linux kernel sense, due to it being a core of this thesis. One must know network stack, to understand how network can be emulated and also to comprehend the implementation of the WEnux emulator, described in chapter 3.

1.1 Network model

International Organization for Standardization (ISO) created an Open Systems Interconnection (OSI) reference model 1.1, which defines a network as 7 separate layers. Core concept of this model is based on abstraction and standardization of different responsibilities of each layer. This thesis mostly focuses on the Ethernet based implementation, along with low-level kernel processes and userspace[1] applications [2].

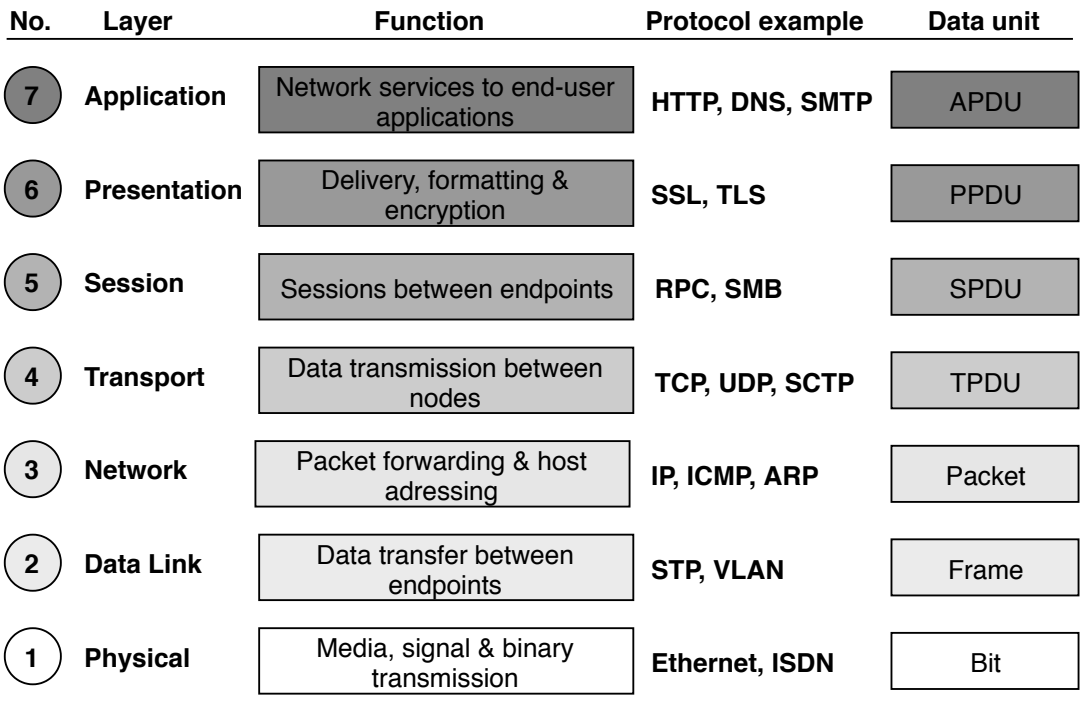

Fig. 1.1: Model of individual ISO OSI layers with their functionality, protocol examples and data units on each layer

[1]Segment of system's virtual memory dedicated to running user applications

Linux kernel stack is handling incoming frames from the second layer upwards to the fourth layer, where it passes the control over segment to the userspace. Kernel handles many network operations on these layers, such as:

- Changes to packet due protocol rules (IP security (IPsec) or Network Address Translation (NAT) rules)
- Discarding the packet (routing not having the entry for given endpoint)
- Handling the network errors and sending error messages (Internet Control Message Protocol (ICMP) sending the control messages)
- Fragmentation of packet (if the packet extends the Maximum transmission unit (MTU))
- Checksum calculation (redundancy check for data integrity)

1.2 Network device

The link layer of Linux kernel is resided by the network device drivers, which are handling the incoming or outgoing traffic to the physical network interface card (NIC). Network device in kernel is represented by important attributes described in the `net_device` structure. Understanding the network device is crucial knowledge to comprehend the network stack[2].

Device parameters can be following:

- **IRQ**[2] **number** - describes the device priority of interrupt request on the CPU
- **MTU** - determines the size of maximum transmission unit coming in or out of the device
- **MAC**[3] **address** - logical hardware identification address
- **Name** - symbolic name containing used to reference the device
- **Flags** - characterizes the state of the device
- **Features** - list of supported features by the device
- **Queues** - numbers of Tx and Rx queues of the device

1.2.1 New API

With the influx of high speed networking the previous implementation of network drivers, based on interrupts for each packet received, was not efficient enough. In order to combat this problem, the new software technique was developed, call the New API (NAPI), based on polling[2].

[2]Interrupt Request

[3]Media Access Control

Interrupt requests - straightforward method of handling each and every packet is to issue and interrupt directly to kernel. However, servicing IRQs is a costly procedure in terms of the CPU resources and time.

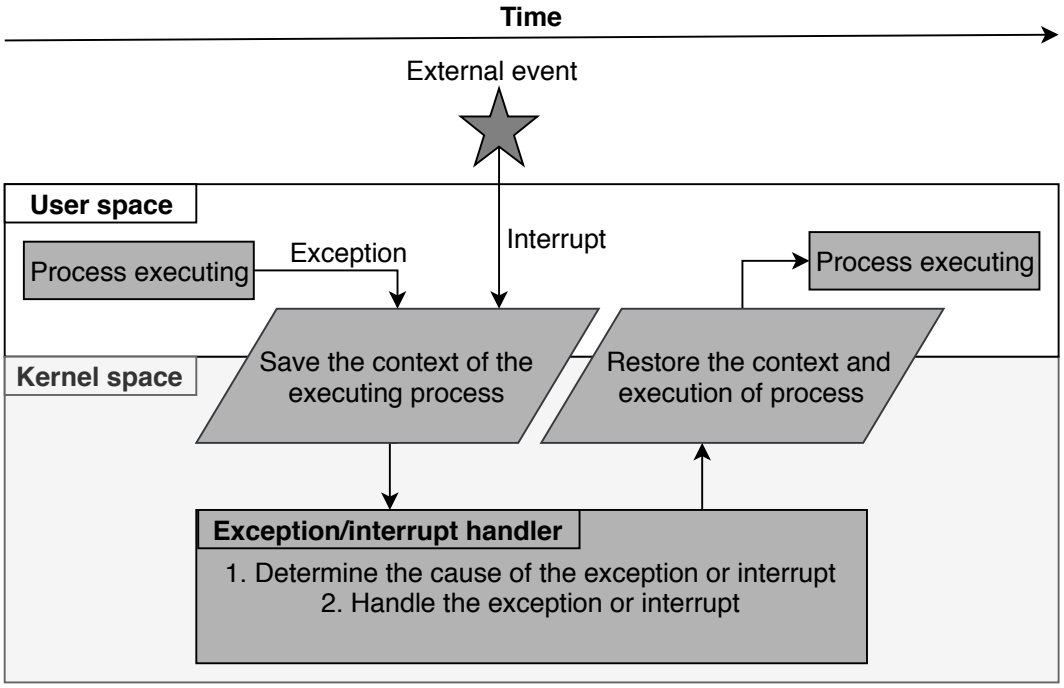

Fig. 1.2: Timeline of interrupt or exception handling [3]

Polling - developed as an alternative to the interrupt-based processing. Kernel periodically checks whether there are incoming packets on the NIC, thus eliminating the need of processing the interrupts. Efficiency of polling is ultimately based on its frequency. Sub-optimal period can either lead to wasted CPU cycles, or on the other side of spectrum to packet loss, due to incoming buffer being full.

As a compromise, the Linux kernel uses the interrupt-driven mode by default and only chooses to switch to the polling mode, when the packet flow exceeds a certain threshold, known as *weight* of the interface.

Result of these two methods is explained in figure 1.3. Whenever a packet arrives, it is added to the network device's receive ring-buffer, and an interrupt is raised on the CPU associated with given memory segment. However, typical network device does have multiple ring-buffers and their respective interrupt numbers are spread across various CPUs. Nowadays, NICs use advanced feature called an adaptive interrupt coalescing (AIC) to reduce the number of hardware interrupts raised during heavy load. This allows the CPU to run uninterrupted for longer time, thus processing more information.

`softirq daemon`[4] is crucial part of the lower parts of the network stack, as it handles the mechanism that copies the network data from the device's memory into a kernel socket buffer. In case of constant interrupts to softirq process from the device, the kernel would fail to move the packets to the internal socket buffer memory, therefore increasing the possibility of packet loss. Softirq process is using NAPI polling, whenever it is trying to dequeue a batch of packets from the NIC, by momentarily masking and disabling interrupts, that might occur [2, 4].

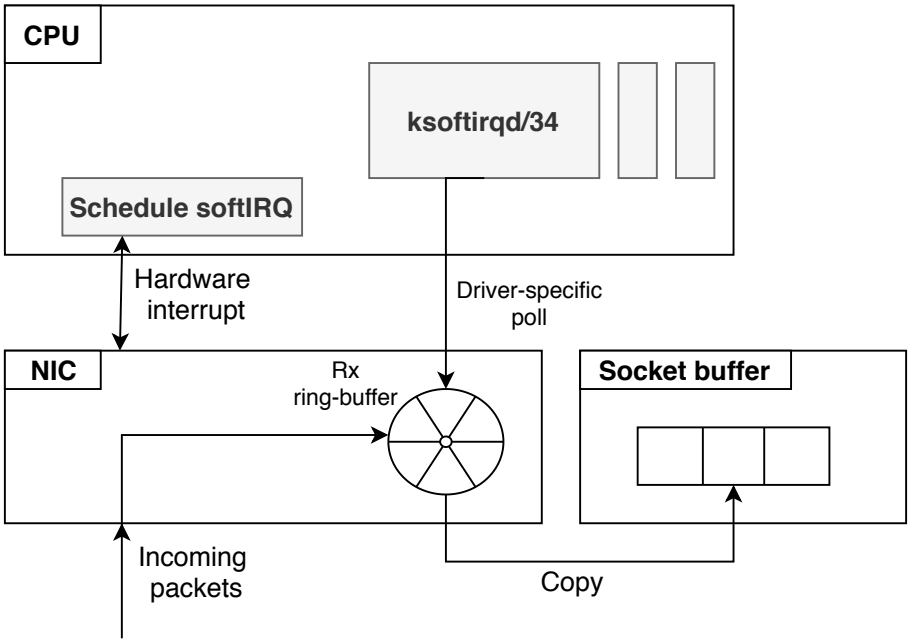

Fig. 1.3: Processing of incoming packets in modern Linux kernel using both polling and interrupts [5]

1.2.2 Role of network driver in kernel

To understand the network emulation, one must know the important role, that network driver plays in it.

Each time kernel needs to send a packet to an interface, first it is enqueued into the queuing discipline (qdisc), configured for that specific interface. Afterwards, the kernel tried is handling the transport of packets between the qdisc and the network driver [6].

Thus main responsibility of the network driver is to:

- Receive incoming packets destined to the host, while passing them upwards to the network and transport layer

[4]Computer program that runs as a background process

- Transit outgoing packets generated on the host and sent to an external network, or to forward traffic received by the host

1.3 Netfilter

In high-level overview, a netfilter subsystem is a framework to register callbacks in different positions of packets in the network stack, also called netfilter hooks, explained in section 1.3.1, and perform diverse operations on packets, such as changes to addresses or ports, dropping them completely, logging and many other modifications. So called netfilter hooks, allow various netfilter kernel modules to register these callbacks to perform numerous tasks on the netfilter subsystem.

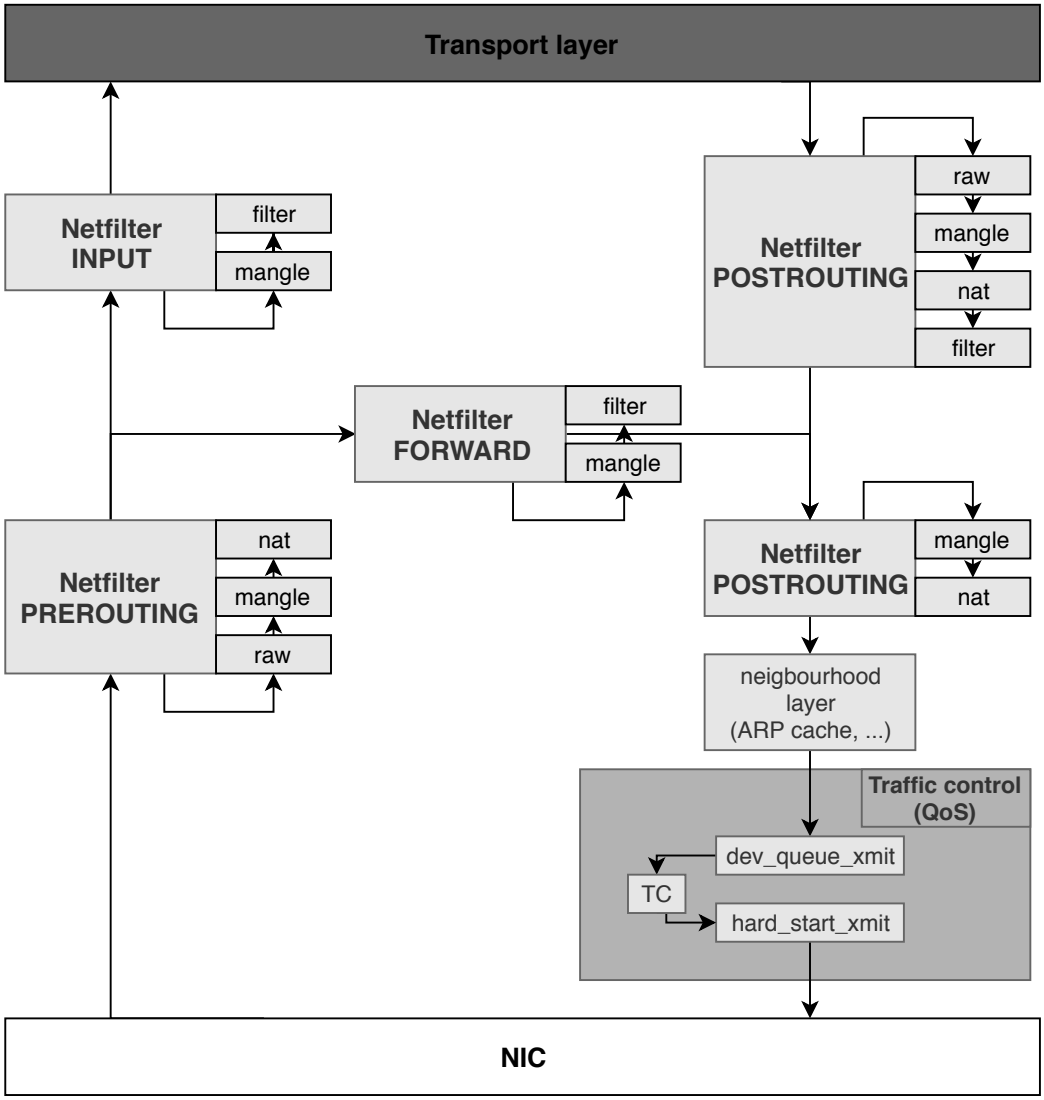

Fig. 1.4: Packet flow in the network layer through the netfilter [7]

As shown in figure 1.4, the netfilter subsystem provides the following functionalities: Packet selection, filtering and mangling, which is mostly executed by the *iptables*, *nftbales* and *bpfilter* modules. Network address translation is also present in netfilter, which is done through NAT table and can be considered as a packet mangling in a way, where it only manipulates the addresses.

Netfilter also tracks connections with the *conntrack*, which keeps a state table to determine and identify connections that belong together through the tracking of IP addresses, ports, etc. NAT facilities also make use of connection tracking, to translate all packets from a flow the exact same way.

Gathering of network statistic is also present as a netfilter module, that is able to log all actions of netfilter frameworks [2].

1.3.1 Netfilter hooks

Hooking is extremely potent tool for monitoring software behaviour or extending functionality without altering the original code. The core concept is to intercept specific system events or calls and use them to initiate custom code.

Hooks can be found in five points of the network stack, shared between both IPv4 and IPv6. These points in packet's traversal through given protocol stack are very well-defined, and netfilter registers these callbacks in order to parse, change and use these packets [8].

Tab. 1.1: Netfilter hooks

Name of hook	**When is hook called**
1. `NF_INET_PRE_ROUTING`	Packet arrives to the netfilter
2. `NF_INET_LOCAL_IN`	Destination of packet is local host
3. `NF_INET_FORWARD`	Destination of packet is other interface
4. `NF_INET_POST_ROUTING`	Packet leaves the netfilter and sent to network
5. `NF_INET_LOCAL_OUT`	Packet is generated locally and destined out

First hook can be found in the `ip_rcv()` function in IPv4, and respectively in the `ipv6_rcv()` function in IPv6. These functions are protocol handlers and also point that all incoming packets reach, before they enter routing subsystem for the lookup.

Second hook located in the `ip_local_deliver()` function in IPv4, and in the `ip6_input()` function for IPv6. All incoming packets, where the lookup in routing

subsystem decided, that they are destined to local host, reach this hook point after first passing through the `NF_INET_PRE_ROUTING` hook point.

Third hook is extremely similar to second hook, which is based in the `ip_forward()` and the `ip6_forward()` functions for their respective protocols. All forwarded packets, meaning that lookup in routing subsystem was a forward decision, reach this hook point after getting here via the `NF_INET_PRE_ROUTING` hook.

Fourth hook is the last netfilter hook, that packet reaches, when it is leaving this subsystem, in his respective functions `ip_out()` and `ip6_finish_output2()`. Forwarded packets, that already left the routing subsystem reach this final point. The `NF_INET_POST_ROUTING` hook additionally serves fifth hook as well, to help packets exit the netfilter framework.

Fifth and the last defined hook is implemented as a stack unique function `__ip_local_out()` in IPv4 and `__ip6_local_out()` in IPv6. Indicated hook is reached when packet is generated locally on the host and later gets to fourth post routing hook to leave the netfilter[2].

1.3.2 nftables, iptables and bpfilter

Another crucial part of netfilter is a kernel module called iptables and its successor nftables. Both of these netfilter extensions have their userspace utilities to configure their kernel counterpart, however, this section only talks about the kernel portion. There is also a third contender for packet filtering, known under the name of bpfilter, which is supposed to soon replace both nftables and iptables[2].

Internals of packet filtering

From higher perspective, iptables[9] and nftables modules allow modification of the kernel module and its parameters through the userspace utility, and for definition of tables containing chains of rules for how each packet should be treated. Each table is associated with different kind of packet processing based on the netfilter hook points. Packets are typically processed by sequentially matching with rules in the chain. Nonetheless a chain can contain a goto or jump rule, which allows for table nesting. However, the main difference between the goto and chain, is that jump rule remembers the point of call, therefore it is bound to return after the nesting rule is explored. As a result of this system, every packet arriving or leaving the Linux host passes through at least one chain.

Figure 1.4 explains the existence of multiple tables, where chains are referenced. Chains consist of rules, where they specify which packets are supposed to get matched. Essential part of iptables rules are so called targets and verdicts, which

can either extend the chain which packet is going to traverse and respectively explicitly make a direct decision, whether the packet should be dropped or allowed through. Packets are matched based on the ISO/OSI model layers, ranging from the data link layer up to the transport layer.

A packet is traversing the chain of rules until either of these events occur:

1. Rule matches the packet and gives the verdict whether to take call `ACCEPT` or `DROP`, or a module that will carry this verdict
2. Rule calls the `RETURN` verdict, therefore packet has to go back to the processing chain
3. Rule does not match the packet and end of the chain is reached, thus traversal continues in the parent chain or in the base chain policy, where the verdict is given

Not only verdict specifies the fate of packet, but targets can also provide verdict, whether to `ACCEPT` the packet, consequently using the `NAT` module, or to `DROP` the packet by calling the `REJECT` module. Main difference between the target and verdict is that it can also rule `CONTINUE`, for example to the `LOG` module, to continue with the next rule as if no target nor verdict was given.

Implementation specifics

These examples are concerning IPv4 network namespace object, known as the `netns_ipv4`. First of all, every table has to be registered with the `ipt_register_table()` or respectively unregistered with the `ipt_unregister_table()` function. After registration namespace object knows a reference to the `xt_table` object, thus the `netns_ipv4` structure also has pointers to the tables, such as the `iptable_filter`, `iptable_mangle`, `nat_table`, etc [2].

To simplify the explanation of iptables and its internal working, example in form of packet flow chart 1.3.2 is presented.

Packet arrives to network layer and is immediately received by the `ip_rcv()` function, which also contains the first hook, packet is going to encounter, in form of the `NF_INET_PRE_ROUTING` hook. However, filter table's callback does not registered this hook, as it is not present in previously mentioned table. Following the chart, the packet leaves the `ip_rcv_finish()` function and goes to routing subsystem 1.4 and performs lookup. In this specific scenario packet is either intended to be delivered locally or forwarded. Routing subsystem, based on its decision, initiates:

- **Local delivery** by calling the `ip_local_deliver()` function, which returns the `NF_INET_LOCAL_IN` hook. Filter table contains this hook, so the `NF_HOOK()` (callback) macro will invoke the `iptable_filter_hook()` function. This filter hook is only an auxiliary function, to call the `ipt_do_table()` function, which

invokes the LOG target callback, known as the `ipt_log_packet()` function. Resulting in the packet header being written to the syslog, as there are no more rules to be called. Thus, the `ip_local_deliver_finish()` function is invoked, and packet's traversal advances to the the transport layer, where it will get handled by its respective socket.

Listing 1.1: net/ipv4/ip_input.c [10]

```
int ip_local_deliver(struct sk_buff *skb)
{
  ...
  return NF_HOOK(NFPROTO_IPV4, NF_INET_LOCAL_IN,
                 net, NULL, skb, skb->dev, NULL,
                 ip_local_deliver_finish);
};
```

- **Forwarding**: by calling the `ip_forward()` function, that returns `NF_INET_FORWARD` hook, similar to the local delivery. This call pattern is intended to be the same, as for the local delivery by continuing with another auxiliary hook call in form of the `iptable_filter_hook()` function, which calls the `ipt_do_table()` function. The packet is once again logged to the syslog, by the `ipt_log_packet()` function. However, at this point the forward table no longer shares its path with the filter table, by having a the `ip_forward_finish()` function in the `NF_HOOK()` argument, which is known as the continuation function. As there is no `NF_INET_POST_ROUTING` hook present, the `ip_output()` function is invoked, which continues to the `ip_finish_output()` function, that hands over the packet to the network driver.

Listing 1.2: net/ipv4/ip_forward.c [10]

```
int ip_forward(struct sk_buff *skb)
{
  ...
  return NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD,
                 net, NULL, skb, skb->dev, rt->dst.dev,
                 ip_forward_finish);
  ...
};
```

nftables

So why was nftables created, if it basically does the same thing as iptables? First of all it simplifies Linux kernel application binary interface and reduces the code duplication, highly present in iptables. As mentioned above, packets are usually inspected sequentially, which is not always true. nftables provides a new implantation over prehistoric iptables, which supports generic set infrastructure, that allows for maps

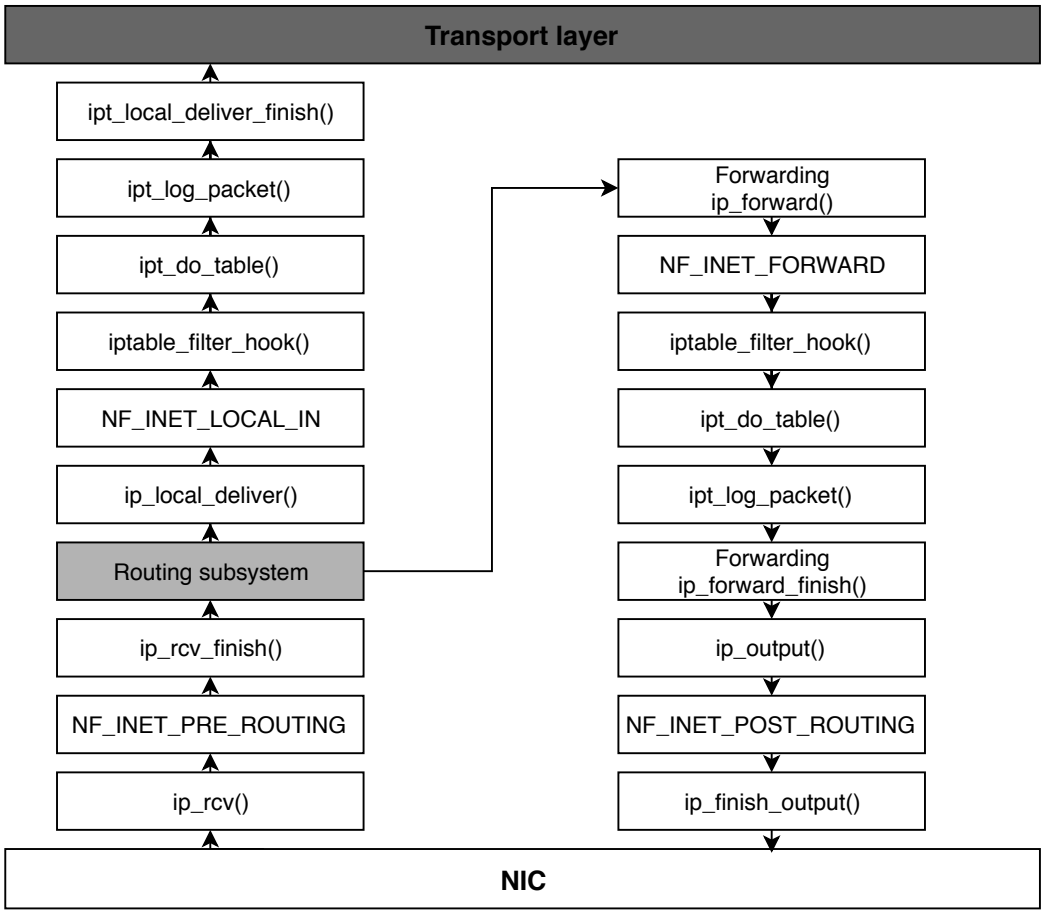

Fig. 1.5: Netfilter's iptables example traversal with forwarded and local destined packets [2]

constructions, as well as, concatenations. This fact alone grants nftables advantage of faster inspection, due to the possibility to create multidimensional tree of ruleset, thus reducing the rules packet has to explore. iptables are also scattered based on protocol used, such as iptables for the IPv4, ip6tables for the IPv6, arptables for the Adress Resolution Protocol (ARP) and ebtables for the Ethernet. Crucial difference is that nftables implementation added a simple virtual machine into kernel that executes the bytecode for the packet filtering. The virtual machines does operations so simple, it is all the way down on the arithmetic, bitwise and comparison logical operations level, with basic packet parameters and its metadata. This virtualization allows for manipulations with multiple sets of data, replacing full set lookup[11].

bpfilter

With the rise of high speed networking the performance of both iptables and nftables is insufficient in case of network speeds higher than 10 Gbits per second. Its extended

functionality also covers high-performance load-balancing, Denial of Service (DoS) mitigation, firewalling, safe instrumentation of kernel and userspace code, etc.

From higher perspective, one notable design difference is that the BPF, does not allow user to tap into raw network, but instead it creates a pseudo-device, acting like a staged controlled area. This pseudo-device is later bound to a network interface and it acts as a network device in the kernel sense, thus reading the buffers directly from the NIC, as well as, writing or injecting new packets to it.

As a result of this implementation is safe and flexible programming environment in various different contexts, such as, networking datapaths, kernel probes, perf events any many more. Previous realization of packet filtering with iptables and nftables added a kernel module, which introduced significant risk, in case something would go bad, which is not the case for BPF programs, as they are verified at load time to ensure no out-of-bounds accesses can occur. BPF is much faster than older iptables and nftables implementation due to just-in-time compilation of its bytecode to the native instruction set of the CPU.

XDP is closely tied to the BPF, as it modifies the packet flow through the kernel network stack and in some cases, it allows the packet to explicitly bypass whole network stack and be directly transmitted to the outbound NIC. Previously, the network datapath was set chain of actions, which is no longer the case with XDP eBPF[5] combination.

To elaborate how XDP eBPF changes the packet flow, we can use 1.4 as a reference. The packet arrives to the NIC and is directly sent to the XDP eBPF, where it decides whether to `XDP_ACCEPT` packet and call the `alloc_skb`, to `XDP_REDIRECT` the packet, thus sending it to the userspace and creating a XDP specific packet with the `sll_family` parameter set to `AF_XDP` or lastly, to bypass the kernel network stack altogether using the operation `XDP_RX` [12].

1.4 Routing subsystem

This section is explaining few advanced ideas of routing, as well as, how routing works in the Linux network stack.

Linux routing subsystem is not only used in the Linux distributions designed to servers or clients, but also systems developed for routers, ranging from SOHO routers, all the way up to the core layer, used in the Internet backbone. This section mainly talks about routing tables, FIB TRIE and much more. TRIE is derived from word *retrieval* and describes a data structure, defined as a special tree that replaced the FIB hash table. Crucial part of routing subsystem is route lookup, how and

[5]Extended BPF, implementation of BPF in Linux kernel

when ICMP Redirect messages are generated and lastly about the removal of the routing cache code.

1.4.1 FIB

Primary responsibility of a routing subsystem is maintenance of forwarding database and forwarding of traversing packets. SOHO networks might have their FIB managed by an administrator, because of the topology being static. However, as we move up the ladder to bigger networks, topology becomes dynamic and maintaining of FIB by hand is rather impossible. In such cases, the FIB is managed by the userspace routing daemons, which alter kernel routing tables, sometimes in conjunction with special hardware enhancements.

FIB table

The core routing subsystem data structure is known as the `fib_table`, which is basically a routing table. Another important part of a Linux routing model is the `fib_info` structure, that contains similar characteristics shared between multiple routes. All of those structures are stored in a hash table called `fib_info_hash`, that is dynamically resized based on the amount of `fib_info` structures currently employed in the routing subsystem.

Maximum amount of routing tables is 2^8, where default tables are known under the `tb_id` 255, for `RT_TABLE_LOCAL` and 254 for `RT_TABLE_MAIN`. The rest of tables are not initialized and can only be utilized with the Policy Based Routing (PBR). Routing table also contains info about the number of default routes in the table under the `tb_num_default` value, as well as, placeholder for a routing entry in form of complete `trie` structure, that points through the `fib_alias` to theW `fib_info`. Between all of those structures are multiple `key_vector` structures, which are either internal nodes or a leaf for trie [2, 13].

Listing 1.3: include/net/ip_fib.h [10]

```
struct fib_table {
  struct hlist_node       tb_hlist;
  u32                     tb_id;
  int                     tb_num_default;
  struct rcu_head         rcu;
  unsigned long           *tb_data;
  unsigned long           __data[0];
};
```

FIB info

This structure is meant to represent a routing entry, but in fact, it often aggregates multiple entries into one.

Listing 1.4: include/net/ip_fib.h [10]

```
struct fib_info {
  struct hlist_node      fib_hash;
  struct hlist_node      fib_lhash;
  struct net             *fib_net;
  int                    fib_treeref;
  ...
  unsigned char          fib_protocol;
  unsigned char          fib_scope;
  ...
  u32                    fib_priority;
  ...
  unsigned int           fib_offload_cnt;
  struct rcu_head        rcu;
  struct fib_nh          fib_nh[0];
 #define fib_dev         fib_nh[0].nh_dev
};
```

It contains important routing entry parameters, such as:

- `fib_dev` - outgoing network interface
- `fib_protocol` - routing protocol identifier of this route, defines how this routing entry was created
- `fib_scope` - identification of destination distance, describes whether the destination is directly attached, locally available or global
- `fib_nh[0]` - specification of nexthop member, defined as array due to the Multipath Routing

Caching

Route lookup can be performed numerous amount of times per second, therefore it is in the best interest to optimize its performance in the routing subsystem. Results of route lookup are typically cached in a nexthop structure, known as the `fib_nh`. However, there are many exception, due to different types of routes, which are beyond the scope of this work.

FIB nexthop

The `fib_nh` structure basically represents the decision of the routing subsystem with all required information to perform packet hand off to the lower layer.

Listing 1.5: include/net/ip_fib.h [10]

```
struct fib_nh {
  struct net_device      *nh_dev;
  struct hlist_node      nh_hash;
  struct fib_info        *nh_parent;
  unsigned int           nh_flags;
  unsigned char          nh_scope;
  ...
  int                    nh_oif;
  __be32                 nh_gw;
  __be32                 nh_saddr;
  ...
};
```

One of the most important parameters it contains, is a direct reference to the `net_device` in form of the `nh_dev` parameter, specifying the outgoing nexthop interface.

Extremely important part of the FIB nexthop are also its exception, which are meant to handle cases, where routing entry has to be changed by the state of network invoked by the ICMP redirect message or Path MTU discovery.

FIB alias

Due to the nature of LPC trie implementation, described in section 1.4.2, it allows for the node reuse and compression by multiple prefixes with similar traits. If route entries differ only in the Type of Service (ToS), FIB generates an alias.

Listing 1.6: net/ipv4/fib_lookup.h [10]

```
struct fib_alias {
        struct hlist_node       fa_list;
        struct fib_info         *fa_info;
        u8                      fa_tos;
        u8                      fa_type;
        u8                      fa_state;
        u8                      fa_slen;
        u32                     tb_id;
        s16                     fa_default;
        struct rcu_head         rcu;
};
```

This means, that there is no need to create the `fib_info` structure for each routing entry, but rather the `fib_alias` which links them together and speeds us the route lookup, as its footprint is smaller than info's. As per of code listing above from the kernel, it aggregates multiple `fib_info` with same parameters seen in the structure.

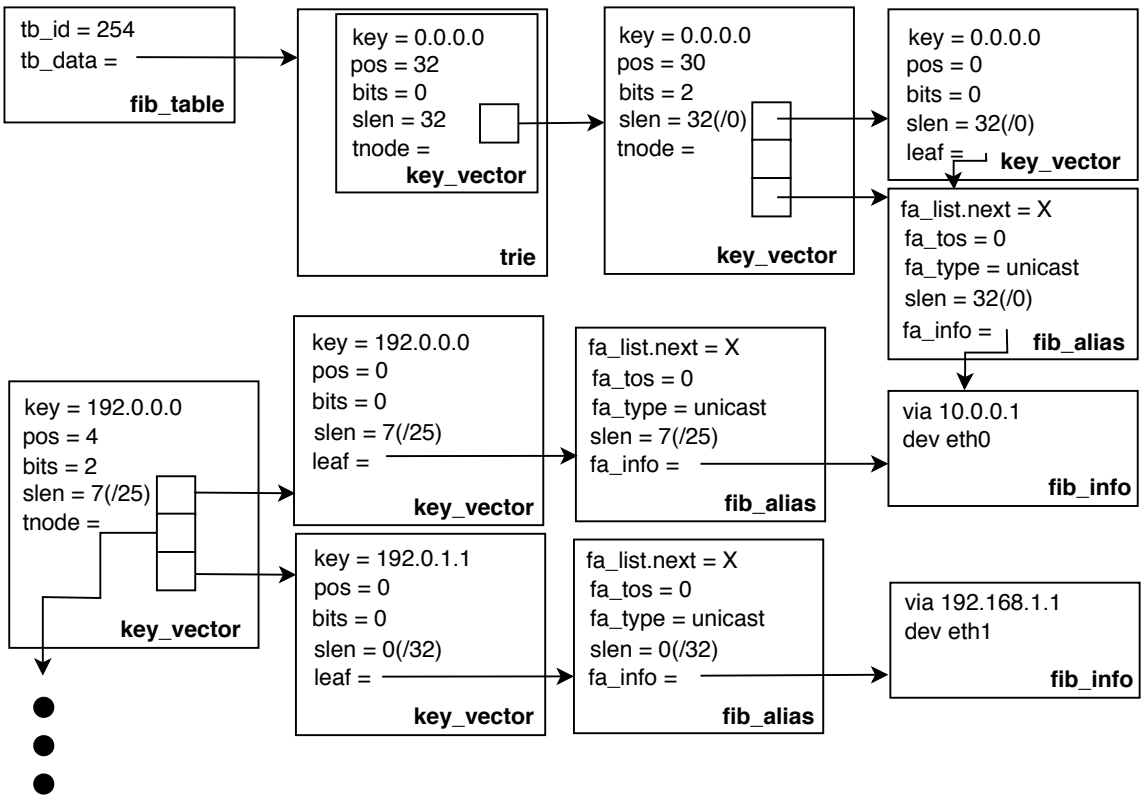

Fig. 1.6: Graphical representation of memory using the LPC trie [14]

1.4.2 Route lookup

Crucial part of being able to communicate over the IP is the route selection to the destination. The selection of a route path is traditionally made on a hop-by-hop basis based exclusively on the destination of the packet. The Linux routing subsystem can be easily compared to a conventional routing device, such as router, multi layer switch, firewall, etc. However, it can be configured to be much more flexible, for example to prioritize routes based on specific packet characteristics.

Trie

Currently the routing subsystem is using the *Level and Path compressed trie*, known as the LPC trie for route lookup. This type of trie detects and compresses densely populated parts of the trie and replaces them with a single node and an associate vector of 2^k children. Such node can handle k input bits instead of a single one.

The LPC trie is a structure that allows *longest prefix match* technique to be utilized much faster than the hash based solution. This technique ensures that

the most specific route will be selected. Use of this method allows routes for large network to be overridden by more specific host or network routes. Conversely, the longest prefix match also allows specific networks or hosts to be aggregated into the larger network address, due to the Classless Inter-Domain Routing (CIDR) [13].

There is also an exception, where destination of the packet does not decide the route, but rather predefined policy, known as the policy-based routing.

Policy-based routing

With the raising complexity of network scenarios, a destination address-based routing is no longer sufficient in those cases, so the PBR takes over. Its native Linux support through the use of multiple routing tables and the routing policy database allows for granual control over the routing path of any packet.

The PBR is an extremely potent tool for the routing, due to its ability of selectors to use any attribute of a packet passing through the routing subsystem. Its attributes can range from the source address, ToS or Differentiated Services (DS) flags, *fwmark* and interface name, where the packet was received, can be used as selectors. Mentioned attributes determine which routing table is going to be used, allowing for higher control over the routing.

When the routing subsystem is trying to determine the outgoing route for a packet, the kernel always used to consult routing cache first. Nowadays, there is no routing cache due to many problems it had, most notable being the DoS.

The routing chain begins to search the Routing Policy Database (RPDB) by the priority. For each matching entry in the RPDB, the specified routing table is searched for the longest prefix match based on the destination address. If the kernel is successful it will select the route and forward the packet. Otherwise, search through the RPDB continues until match is found or all of the sub-routing tables are exhausted [15].

Implementation specifics

To begin this subsection, one must know, that each packet triggers lookup in routing subsystem, in either Rx or Tx path.

Route lookup for the destination address begins with the `fib_lookup()` function. When it finds a proper entry in the routing subsystem, it builds the `fib_result` object, consisting of different routing parameters. The lookup function call contains important `flowi4` or `flowi6` object respectively, which stores packet specific attributes, such as the destination address, source address, ToS, etc. Based on this object the key for lookup is defined and it should be initialized prior to the function call. The `fib_lookup()` starts its search in the local FIB table. If the lookup

succeeds, the `dst` object is built, containing the `dst_entry` structure and the destination cache. This object is embedded into a `rtable` structure, which is known as the routing entry with reference to the SKB. The `dst_entry` is utterly important in routing process as it contains two callbacks named the `input` and `output`, where only these two can manipulate with the SKB.

To understand how route is propagated through the kernel, there is need to delve deeper into the attributes of the `rtable`. First off, the `rt_flags` parameters contains flags, describing the type of destination address, whether it is a broadcast, multicast, local or redirect address. The `rt_uses_gateway` parameter defines if nexthop is gateway or a direct route. Last important parameter is the `rt_iif` specifying the `ifindex` identifier of the incoming interface.

Next step in the routing subsystem is to determine where the input callback is going to be set.

- incoming unicast packets with local host destination - `ip_local_deliver()`
- incoming unicast packets with remote destination - `ip_forward()`
- locally generated packets with remote destination - `ip_output()`
- multicast packets - `ip_mr_input()`
- filtered packets by the kernel or user - `ip_error()`

Last step to decide the result of routing is known as the structure `fib_result` with following important parameters specifying the outgoing route. It contains a pointer to the `fib_info` object, that holds a reference to the `fib_nh`. Another crucial pointer is to the FIB table, as well as, reference to the `fib_alias` list, containing all objects associated with given route [2]. Relations between these structures can be seen in figure 1.6.

Summary

This section described the routing subsystem in detail, with Linux implementation specifics, how are the incoming and outgoing packets handled and organization of memory regarding the FIB routing structures. However, various advanced routing topics, such as the multicast, multipath and PBR, in detail, are beyond the scope of this work.

1.5 Traffic control

This section will not only talk about the specifics in the Linux network stack, but also explore this topic in general sense, as it is the core concept of the thesis.

Quality of service, or so called traffic control (TC) is the name given to the sets of queuing systems and mechanism by which packets are received and transmitted

on a device. TC is one of the last segment of the Linux network stack before packets are sent to the network driver or one of the first segments when incoming packets are handed over to the network stack from the network driver. It includes a decision regarding the incoming and outgoing packets, whether to accept them at what rate, order, etc. For the purpose of this work, policies are created from given set of traffic control systems, which are meant to cripple or degrade the network in specified way, to emulate the WAN.

Queue organization is essential part of packet-switched network due to their nature of being stateless. Statelessness is one of the fundamental strengths of IP networks, where different types of flows are not differentiated. This alone, creates a requirements, where serialized outbound data have to be managed by a queue.

One of the main advantages of traffic control is more predictable network usage and less volatile contention for network resources. The network is set to meet the goals of the traffic control configuration, that has been communicated with users. This policy is known to users and they know what to expect from the network.

On the other side of the spectrum traffic control is extremely complex discipline. Not only in terms of sheer knowledge required to properly configure the policies, but also the computing resources, which can be quickly exhausted or over utilized, as the system has to manage all the structures required for handling the traffic control[16, 17].

When a packet is sent in the egress path, the TC is going to be applied in three phases in the following predefined order, if they contain user-defined policies: **Classification**, **Scheduling** and **Shaping**. A packet coming in the ingress path will only pass through **Policer**. There is also a possibility for the ingress path to apply other types of traffic control, explained in section 1.5.7.

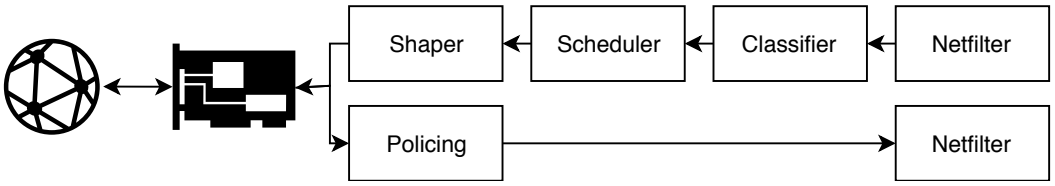

Fig. 1.7: Sequence of different algorithms applied during the packet's traversal [17]

Not necessarily every ingress or egress path contains marking or dropping actions, but they are certainly important parts of traffic control. **Dropping** discards either an entire packet, flow or even classification. **Marking** is a mechanism by which a packet is altered in a known predefined way.

1.5.1 Flows

A flow is defined as a sequence of packets from a distinct source to a distinct unicast, anycast or multicast destination, that the source desires to label as a flow. It can contain all packets in one direction in a specific transport connection or a media stream, but it does not necessarily map a transport connection in a 1:1 ratio [18]. A flow is describes as a 5-tuple consisting of packets with the same:

- source address
- destination address
- source port
- destination port
- transport protocol

One of traffic control mechanism is ability to separate the traffic into classes of flows, known as the classifier. Flows can be aggregated later in the TC pipeline and transmitted together. Another approach is to divide the bandwidth equally based on the individual flows, but this might be extremely challenging task, as each flow competes against one another and some applications might build a large number of flows on purpose.

1.5.2 Tokens and buckets

Another crucial part of quality of service, based on tokens and buckets, is known as the shaper.

Dequeuing can be done multiple ways. Simplest way is to create a counter of packets being dequeued as each item is taken from the queue, which results into complex usage of timers and measurements to limit this rate accurately. Smarter and indeed used way in traffic control, is to generate a tokens at a desired fixed rate, and only dequeue packets if a token is available. Generated tokens are stored in a bucket with fixed size, where they can pile up, if they are not needed, up to a bucket size, and then be used in a burst. This is core concept of rate-limiting or shaping, to support such bursty traffic as HTTP.

To bridge this with the Linux kernel, Token Bucket Filter (TBF) qdisc is an example of a shaper. It works on the exact same way as previously described token and bucket combination. The TBF generates `rate` tokens per second and `burst` defines the size of bucket. Regular traffic, that is highly predictable can be easily handled by smaller bucket size. However, burstier traffic needs larger buckets to be satisfied, unless the burstiness reduction of the flow is required.

Combination of tokens and buckets is used in many qdisc, such as specified classless TBF and classful Hierarchy Token Bucket (HTB).

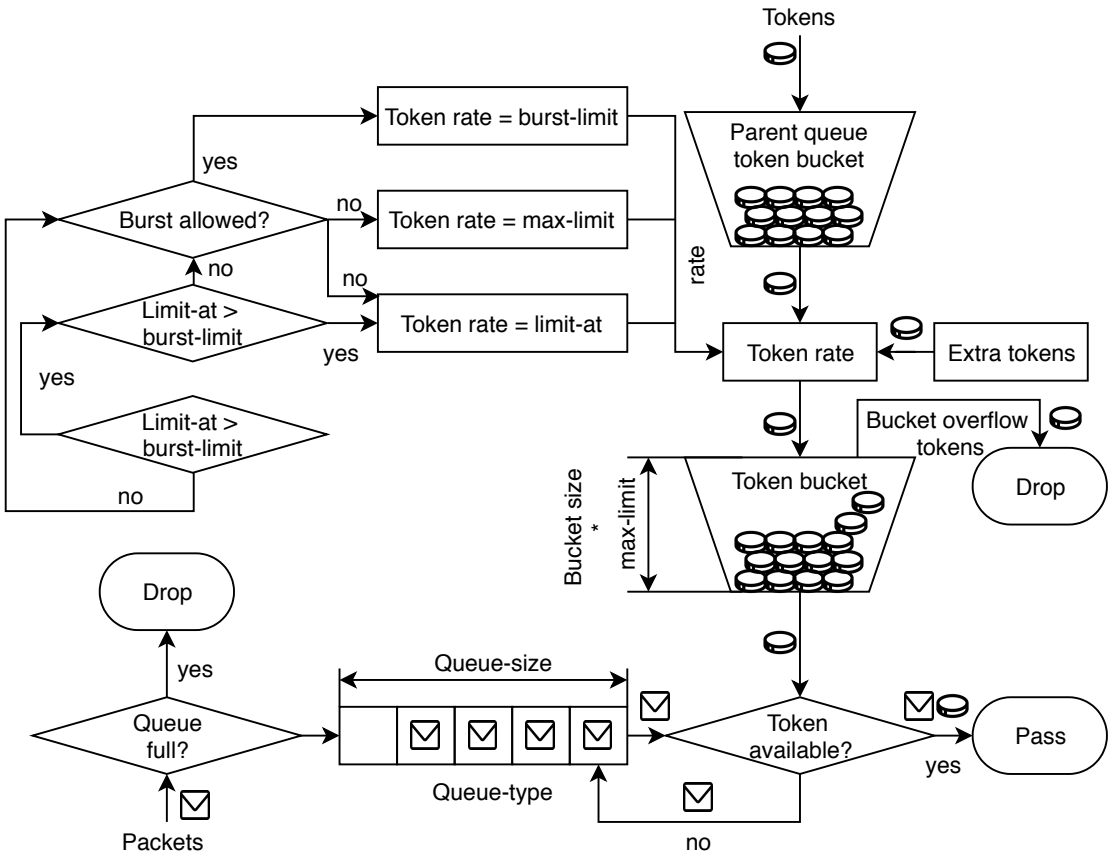

Fig. 1.8: UML diagram of relations between qdiscs and classes [17]

Figure 1.8 describes the activity diagram of a token bucket, where multiple values in diagram are abbervations standing for:

- `limit-at` - guaranteed upload/download data rate to a target
- `max-limit` - maximal upload/download data rate that is allowed for a target
- `burst-limit` - maximal upload/download data rate that is allowed for a target while the burst is active

Maximal token rate is defined at any point in time as the highest of mentioned values, with an exception to burst-limit, that is only active in the burst state. In case where limit-at is the highest value and extra tokens need to be issued to compensate for all missing tokens that were not borrowed from it's parent queue [19].

1.5.3 Classification

Classifier is meant to sort or separate traffic into queues, by identifying individual flows and determining which policies to use.

Classifying is the mechanism to separate packets to be segregated and treated differently, possibly putting them to different egress queues. All of this is based on

network flows, which are distinguished based on the metadata and packet headers, allowing relevant policy to be dispatched.

Such identification can be done through the traffic control by itself, conntrack or newly by eBPF. TC classifier's implementation is the simplest of the bunch, but also offers the least options. It can classify packets mostly based on metadata of the individual packet and there is also a possibility to extract certain information from the packet header. The conntrack is much more complex system, which can be used to divide group of packets into flows, track flow's state and more. This classification can simplify application of specified policy, which can be applied on an aggregated flow, instead of individual packets.

Classification can also include packet marking, which happens on a boundary of a single traffic control domain or it can be done on each hop separately.

1.5.4 Scheduling

Scheduler typically organizes and sometimes reorganizes packet order meant for the output. As it turns out, the main part of this section is a queue management, specifically how are packets sent and how they are treated when the queues are full.

Scheduling is the mechanism to arrange or rearrange packets between the input and output of a specific queue. Most common scheduler by far is also one of the simplest, known as the First in First Out (**FIFO**) scheduler, which does not compensate for various network conditions and does not prefer any flow in a default state. When granular traffic control is required, other generic scheduling algorithms are mostly used, for example:

- **SFQ** - Stochastic Fairness Queueing is a fair queuing algorithms that attempts to prevent any single client or flow from network usage domination.
- **WRR** - Weighted Round Robin is a round-robin algorithm which gives each flow or client a change to dequeue packets.
- **RED** - Random Early Detection is congestion avoidance algorithm mostly utilized in backbone.

1.5.5 Shaping

Shapers are meant to delay packets and release them at a disciplined pace to enforce policies such as rate limits.

Shaping is the mechanism of packet delay in an output queue before a transmission to meet a desired output rate. Most common use case is to limit the bandwidth and smooth out spikes in the network. Shaping mechanism can be viewed as a work-consuming operation, where it requires resources in order to delay packets. Thus, queuing mechanism performing shaping function is a work-consuming operation.

A work-conserving queuing mechanism such as *PRIO* is not capable of delaying a packet.

One of the main advantages of bandwidth shaping is the ability to control latency of packets. Shaping algorithms are typically a variation of previously mentioned token bucket algorithm 1.5.2, allowing it to control the rate.

1.5.6 Policing

Policers measure and enforce predefined bandwidth and burst limit in a particular queue. They can be found in an ingress path of packet and perform very similar function as shapers 1.5.5, however the key difference is not only in the packet's flow direction, but in their respective actions. Policers can only take on a single corrective operation, known as dropping.

Policing is the mechanism by which traffic can be limited, in the sense of traffic control. Its most frequent usage is to keep the peer in check regarding its bandwidth usage, on the network border. A policer is going to accept traffic until a specified rate, and then drop all the traffic exceeding this limit. Nonetheless the traffic does not necessarily have to be dropped, but it can also get reclassified. It is an extremely simple mechanism to control an ingress queue state. If the packet has arrived to NIC and has been scheduled to enter the queue, its rate is compared with predefined limit and depending on the result, it is either allowed to enqueue or not. Although the mechanism seem simple from the black box perspective, internally it uses token bucket implementation, but does not posses capability to delay a packet as previously mentioned.

1.5.7 Components of Linux traffic control

Described general elements of traffic control can be directly interconnected with the Linux implementation.

Tab. 1.2: Correlation between traffic control in general and the Linux sense

Traditional element	**Linux component**
Classifier	`filter` identifies as a part of `classifier` object
Scheduler	`qdisc`, can also contain classes and other qdisc
Shaper	`class`
Policer	Exists as a component of a `filter`

filter

`Filter` can be described as one of the most complex component in the Linux traffic control. Its implementation provides a mechanism, which aggregates multiple key elements of traffic control. As per table 1.2, most obvious role is to classify packets into an output queues with use of a single or multiple filters defined by the user. Filters cannot exist on their own and have to be attached either to classes or classful qdiscs. However, enqueued packets will always enter the root qdisc and after its traversal, the packet may be directed to any subclasses, where further classification can be performed.

Policer can be considered as a part of filter, where it calls one action above the threshold and other action below the specified threshold. Result of this implementation can lead to a simulation of three-color meter.

Arriving packets are classified as either green, if they fully comply to the requested guaranties, yellow if they are between the minimum guaranties and peak threshold, or red if their values are not acceptable. Such implementation in the network congestion allows red packets to be dropped before yellow ones and so on.

Most notable fact about policing, is that it is not physically capable of delaying packets, as it does not have an access to any queue, thus packets cannot be dropped in a sensible way. This leads to random packet drops, whenever policy is violated, later resulting in a bad behaviour in protocols such as TCP, where retransmission timeouts are going to get triggered, thus introducing latency issue when multiple retransmissions occur.

qdisc

The Linux scheduler is know an `qdisc` in the kernel. Whether user wants it or not, every interface does have a scheduler of some kind and Linux uses a FIFO by default. This scheduler is the simplest implementation possible, which does not interfere with traffic order nor contain any rules. To be exact the default qdisc is `pfifo_fast`. It is based on a conventional FIFO qdisc, with added prioritization. It provides three different bands, known as the FIFO queues for separating the traffic. The band 0 is filled with highest priority traffic and always serviced first. By analogy the band 1 is emptied of pending packets before the band 2 can be dequeued [10, 16].

A queuing discipline can be divided based on its relation to classes.

- **Classful qdiscs** can contain classes and provide a handle for filters to attach to. They can be also used without child classes at the expense of resources, without added functionality.
- **Classless qdiscs** cannot contain classes and and do not provide handle, which makes it impossible for filters to attach. They are used to accept data and

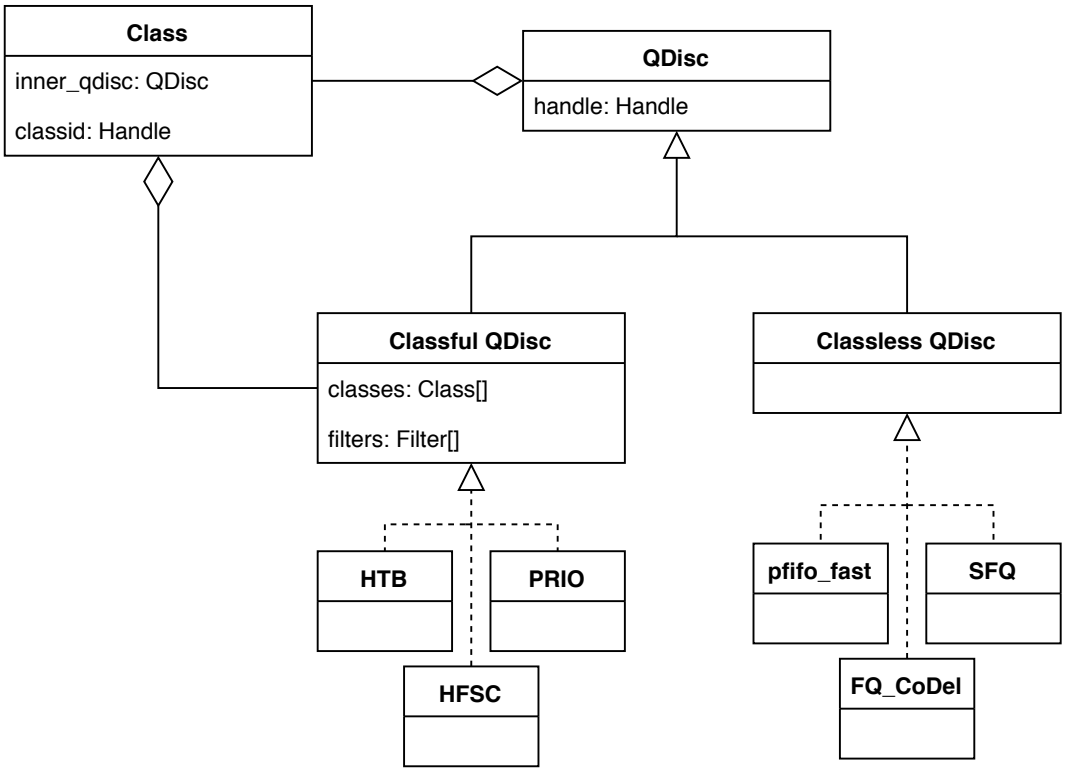

Fig. 1.9: UML class diagram of relations between qdiscs and classes [17]

only reschedule, delay or drop them, without any subdivisions.

Each interface defines a location onto which a traffic control structures can be attached for either an egress or ingress path. The most used is the egress qdisc, known as root dqisc, which is able to contain any other qdisc and class structures. It allows for an extended configuration and creation of the child classes. Therefore, outgoing traffic is always passed through the egress or root qdisc.

On the other side of spectrum, incoming traffic is handled by the ingress qdisc. Compared to the root qdisc, it comes with limited capabilities, where no child classes are allowed and it only exists as an object, that provides a handle for the filter. The ingress qdisc is mostly used as a convenient object for policer to limit the incoming bandwidth.

In conclusion, the egress qdisc is much more useful in overall traffic control scheme, as it contains a real qdisc with possibility to classify, schedule and shape the traffic, as opposed to the ingress qdisc that can only apply specified policies.

class

Classful qdisc, such as the HTB, is a predisposition for `class` existence. Classes are known in the Linux as a form of shaper and offer great flexibility, as they can contain

either single child qdisc or multiple children classes. To increase the complexity of traffic control scenarios, one can also define a classful qdisc itself, inside a class.

The amount of filters attached to a class is not limited, which allows for the selection of a child class or for the use of a filter to reclassify or drop traffic entering an individual class. Inner class or root class is defined as a class that contains a child class. This hierarchy is terminated in a leaf class, which ends the chain by containing a qdisc but never any child class [16].

IFB

Intermediate Functional Block is a virtual network device with inverted egress and ingress paths allowing to shape incoming traffic by attaching arbitrary qdisc to the IFB. Device creation is tied to an ifb kernel module, which has to be loaded. After the IFB has been created, an ingress traffic can be redirected from the physical NIC directly to the newly created device. The conntrack can also play a role here, by calling the `CONNMARK` target to restore the connection mark which has been modified by the flow redirection [20].

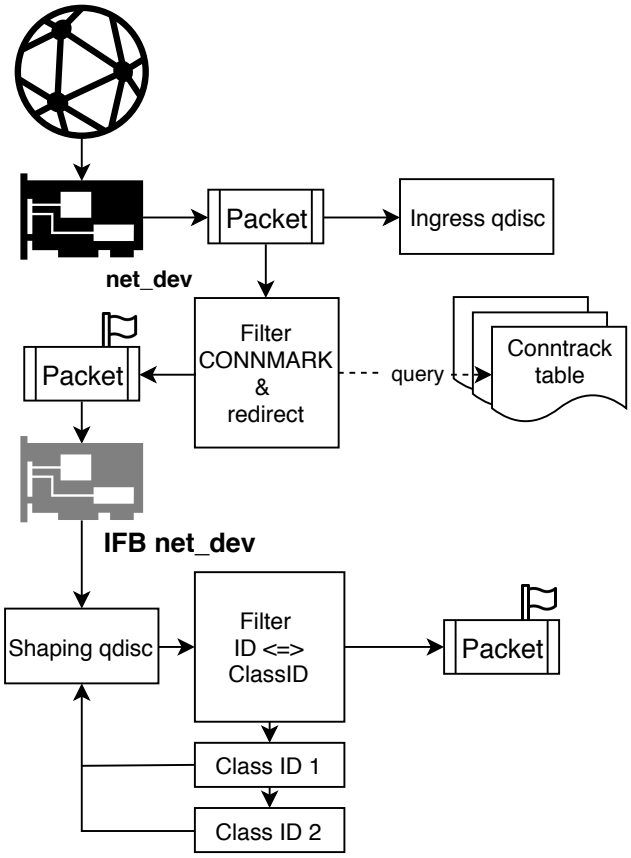

Fig. 1.10: Ingress data path with IFB network device [17]

2 Virtualization of network

This chapter elucidates the virtualization of network and closely explains concepts of simulations and emulations. Since the main focus of the thesis is an emulation of transmission network, the section elaborating on the emulators is much more extensive. However, there is a need to explain the correlation between the model and simulation as well, to better understand the virtualization as a whole.

2.1 Models

A model can be described, as an object that aims to re-create something from the real world. Specifically computer model can be represented as a set of algorithms and equations used to capture the behaviour of the system being modelled. There are many attributes on which one can categorize the computer models, most notable being: time, entropy and dynamicity [21].

- **Continuous model** represents continuous data, where they have a potentially infinite number, divisibility and attributes. Their best description can be given through differential equation.
- **Discrete model** is analogue to continuous model. State variables can only take on a countable set of values, such as integers. Discrete models define the smallest unit of time supported, and events may only occur at discrete timeline.
- **Stochastic model** allows for random variants in variables over time, which serves as a primary tool for estimation of probability distribution of potential outcomes of the system.
- **Deterministic model** is a model, in which the values for the dependant variables of the system are entirely determined by the parameters of model. In contrast, stochastic, or probabilistic models introduce randomness, best represented as probability distributions.
- **Dynamic model** is dependant on time, allowing for interactions between variables over time, as well as, examination of relationships, that could not be sorted out otherwise. A valid example can be a computer network running over several hours, where the state of each network component varies based on traffic generated.
- **Static model** is described as, invariant to the course of time, can be considered to only work with snapshots[1] of the system. Model represents a system at a given point in time or compares the system at different points in time.

[1] State of system at a given point in time

2.2 Simulators

Due to the thesis being focused on emulation, this section will only elaborate on network simulators, to understand the difference.

Simulations are based on different types of models described in section 2.1. Every model leverages some strengths and weaknesses when it comes to the network simulation.

First off, network simulator, compared to a real network, is implemented only as a software program modelling the behaviour of a network by calculating the interaction between different routing entities. This fact, forces any network simulator to implement all parts of network, ranging from electrical signals, protocol stack to network appliances. Depending on the network under simulation, various parts of network have to be simulated, while others can be omitted.

Typical network simulator is a dynamic DES, where events are scheduled dynamically with passing time. On the other side of spectrum are static Monte Carlo based simulators, where repeated sampling is used to compute the result [22].

Network simulators are commonly used in research, engineering and development of the various parts of network. *Cisco* is well known for its *Packet Tracer* simulator, mostly used for education, as its support of protocols is limited and proprietary restricted. For professional use in engineering and development, common simulators are *GNS3*, *EVE-NG*, *NS3* and others.

2.3 Emulators

An emulator is a piece of hardware or software, which is aiming to duplicate (or emulate) the functionality of a specific system, it models, resulting in a replacement of the given system.

Emulator's main goal is to reproduce the external behaviour of the original system, which can be viewed as a black box[2]. Therefore, it has to accept the same type of input parameters and produce the same output as the real-world counterpart. For the external observer it has to look like as if the emulator is performing the same function as the original system. However, the implementation of the given function does not have to reflect the real-world system, it just has to behave live the original function[1].

One can describe emulator in a way, where one computer called *the host* is enabled to behave as another computer described as *the guest*. In this sense, it allows

[2]A device, system or object which can be viewed in terms of its inputs and outputs, without the knowledge of internal workings

the host system to run software or use peripheral devices designed for the guest system. Good example can be obsolescence, where new versions of Windows, operating system, drop support for specific functions from older releases and functionality of applications using these obsolete functions are left to be emulated, instead of rewriting the code to use a newer functions or any other type of maintenance. As another valid example, can be a fact, that many computer printers are designed to emulate Hewlett-Packard LaserJet printers, due to the sheer amount of the software written for HP printers, thus making it easier to develop the printer itself, without much need to care about the accompanying software[23].

Well-known computability theory or so called Church-Turing thesis[24], implies that any operating environment can be emulated within any other environment, assuming that enough memory is available. In reality, it can be extremely difficult to replicate the exact behaviour without proper documentation, hence reverse engineering has to take place. However, that it not the main concern with this theorem in practice, where the timing constraints exist. If the emulator speed does not match the original system, either by running on worse hardware or slower implementation, it might run to timer interrupts, possibly altering behaviour.

To get to the core of this thesis, we need to explore the network emulators. Due to the native Linux environment, traffic control along with its enhancement Netem are great examples of network emulators. Netem[25] allows for modifications of delay, packet loss, duplication and many other network traffic properties, to emulate characteristics of a wide area network. Netem, in black box sense, is acting as an actual network to external systems.

3 Emulator design and implementation

WEnux stands for WAN Emulator for Linux. It has been developed as an open software by Jozef Urbanovský, under the GNU General Public License v3.0.

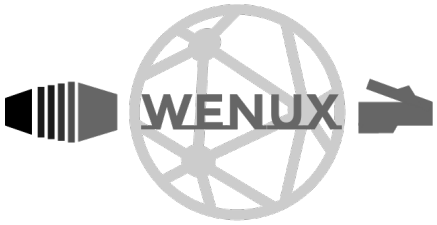

Fig. 3.1: Logo of WEnux

3.1 Research of existing technologies

As an inspiration for WEnux, was software emulator called *WANem* [26], which is frankly over engineering its use case and makes itself seem way more complicated than it should be. WANem is no longer developed or maintained, but it does present a good concept how network emulation can be performed on Linux machine.

WANem was more or less researched from the user perspective, how does it present itself and what are its capabilities. However upon further inspection, it is mostly written in a Bash, with usage of userspace calls to control selected kernel modules.

WEnux was implemented before the code inspection of WANem happend and there was a revelation that WANem does emulation on almost equal terms as WEnux. Both are using *iptables*, *iproute* and ultimately *tc* utilities from userspace, to manipulate kernel parameters in *netfilter*, *conntrack*, *tbf* and *netem* modules. WEnux in comparison to WANem is written in much more sensible way, where it uses object oriented approach in Python3 and code is easily manageable and extensible at any time. On the other hand WANem internals are written mostly in Bash with PHP frontend.

3.2 Experimenting and prototyping

Due to native Linux environment with full access to the system, there is no need to reinvent the wheel, by writing everything from a scratch. It makes a lot of sense to use already predefined kernel modules dedicated to this purpose, such as netem and its userspace configuration utility tc.

First experiments were conducted with basic configuration of *tc-netem* to achieve delay with egress *qdisc*. This was done manually to ensure, that tc is a viable way to proceed. After successful tests with regards to delay, jitter and loss, decision regarding programming language had to be made. There were no constraints from instructions and speed of emulation would fall on kernel itself, making the implementation independent from the programming language.

Python as a candidate sounded suitable for work of this kind as it was more or less supposed to provide a wrapper around userspace utility and possible extension to frontend interface later on. As an extremely versatile object oriented language, this pick turned out right.

At the beginning of the WEnux, the code itself was written in a procedural structural way, without employing any of Python's object oriented strengths. As a proof of concept it was sufficient and tested with regards to *iperf3* generator, *ping* and http speedtest. Egress qdisc was first to be supported with binding to a specified NIC, as well as, ability to include or exclude network and transport layer identifiers. Egress qdisc was extended with netem to manipulate outgoing packet attributes, such as loss, duplication, jitter and delay. Soon after support for tbf was added with ability to limit the rate, size of token bucket itself and latency time, for which packets can stay queue before being dropped.

For simplification of ingress traffic IFB has been created to handle incoming traffic instead of a filter that would serve as a policer. This allows ingress traffic to be redirected to IFB, and be treated as if it was egress traffic. This allows more code reuse in a form of shared configuration of the qdisc hierarchy.

First released implementation of WEnux was using command line interface for control, with extensive argument parser to handle all the possible configuration options. This result was submitted as a semestral thesis. It was able to attach itself on an interface and emulate WAN-like conditions for passing traffic. It was able to manipulate almost every parameter, that made sense to support, in netem and tbf submodules in tc.

Implementation has been separated to `Tc` module, which contains code related wrapper of tc, netem and tbf. There was also a `Utils` module that handles processes unrelated to networking. Entry point to WEnux was through `wenux.py` script that contained an argument parser for all implemented parameters. It also handled signals to allow for easier control of application.

3.3 Final application

WEnux was designed from the start, to be extended of graphical user interface, to allow for easy configuration and interaction with user. This section discussed the final form of application, that is being submitted and decisions that lead to it. Code itself is documented with *docstring* style comments, with only some parts highlighted here.

3.3.1 Graphical user interface

As WEnux is meant to run on Linux system, it was not required to think of portable solution that could run on a server or on any other system. As previously mentioned WANem had front-end implemented as a web interface, with its own server, running remotely or locally. As WEnux is not trying to be as robust to run remotely, it did not make sense to pick web interface along with web server, which could possibly take even more resources from emulation itself. To also allow for easy maintenance and minimal knowledge of different programming languages Python GUI framework made the most sense.

Main contenders for GUI framework were *PyQt5*, *PyGUI*, *Gooey*, *PySimpleGui* and *Tkinter*. In research phase PyGUI and PySimpleGui were favorites, due to being lightweight and easy to work with. PySimpleGui is actually a wrapper for Tkinter and Qt and allows user to pick between these to frameworks in a minimalist way. Tkinter was considered due to being lightweight object-oriented layer on top of Tk. It has the advantage of being included with the Python standard library, making it the most convenient and compatible toolkit to program with. PyQt5 is one of the most widely used framework for GUI development, developed under GPL license, offering robust tools, that can not only serve for creating user interface. PyQt5 also offers Qt Designer, which is extremely convenient builder of graphical user interface that uses WYSIWYG editor. On the other hand Gooey is not as well known framework, but it builds upon programs that utilize argument parsers.

At the beginning there were experiments mostly with Gooey, as WEnux already had quite robust argument parser, which Gooey was trying to create a GUI out of, but it wasn't configurable enough in some cases. PyQt5 with its Qt Designer quickly replaced Gooey as preferred framework for development of GUI.

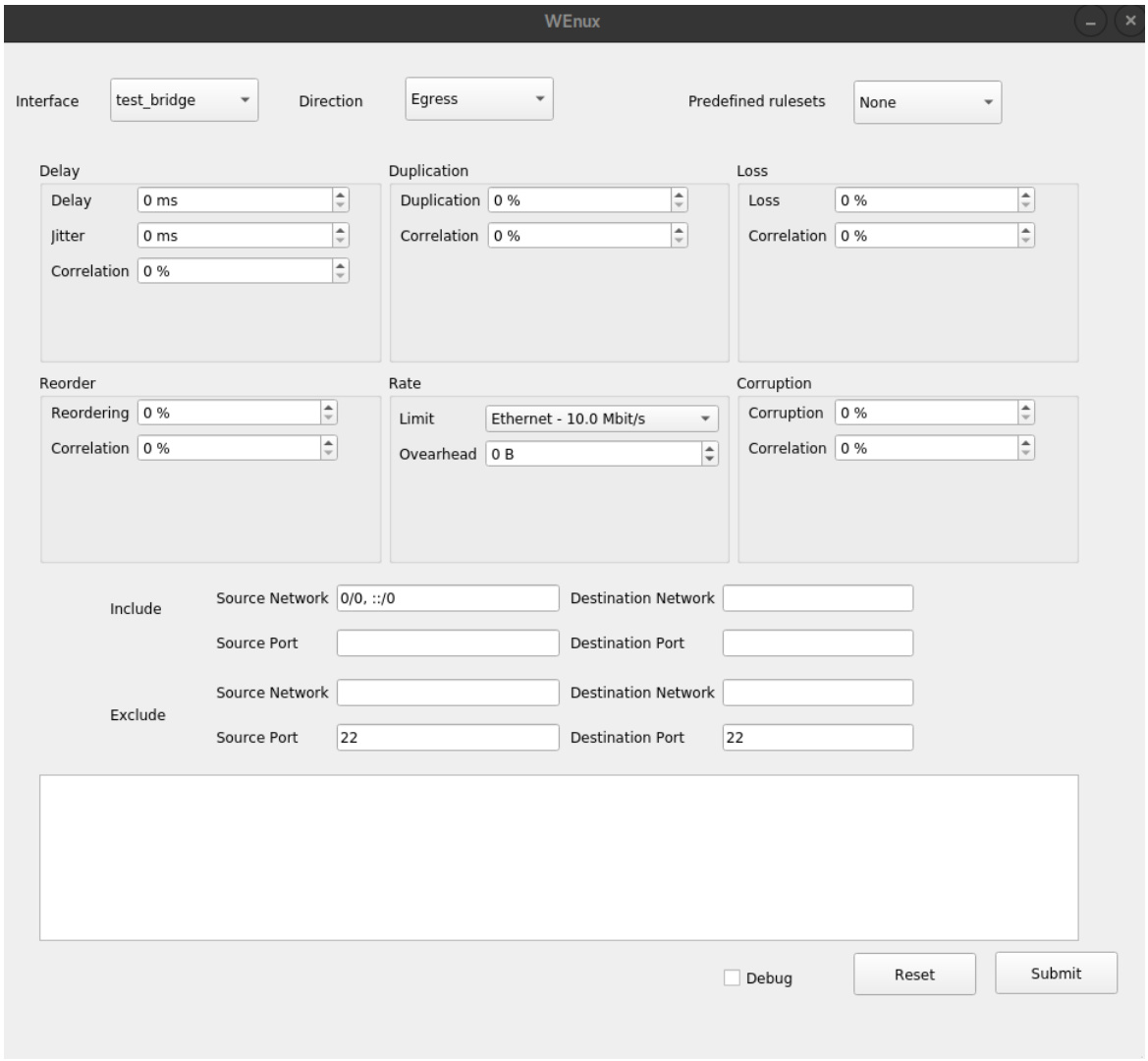

Fig. 3.2: Fresh start of WEnux application

PyQt5 and Qt Designer

PyQt5 is a library, that lets user utilize *Qt GUI framework* from Python. Qt itself is written in C++ and is wide spread cross-platform framework for development. Such an approach allows for easy and fast development under Python, while using all of the advantages of C++, especially its speed, being compiled language. PyQt5 offers not only graphical user interface API classes, but also XML handling, network communication, regular expressions, threads, SQL database, multimedia and other technologies available in Qt. All of those classes are contained withing a top-level Python package called PyQt5, which is required to run WEnux, as well.

Main components used from PyQt in WEnux are:

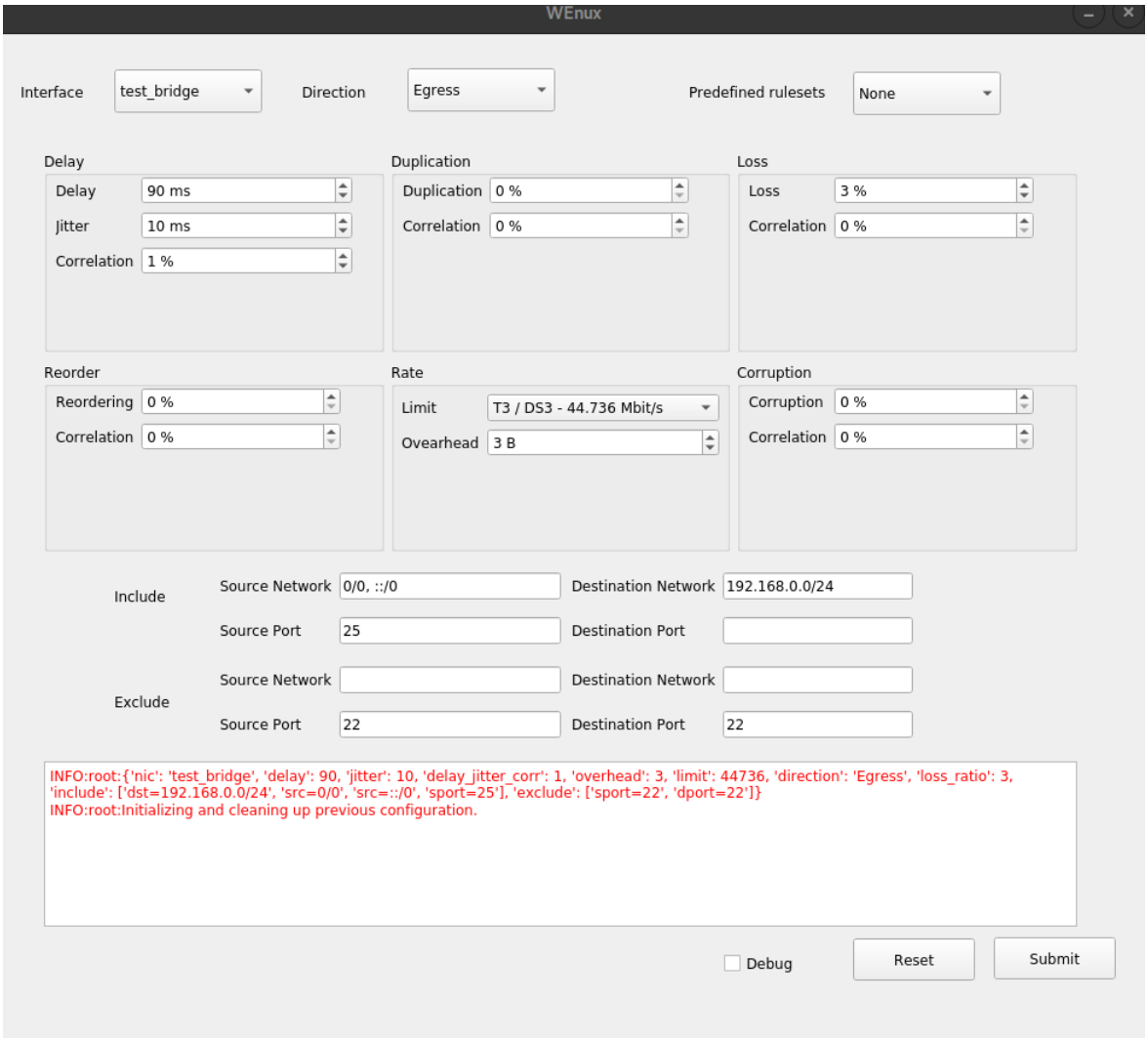

Fig. 3.3: Sample configuration and execution of WEnux application

- *QtCore* module contains the core non-GUI classes, mainly used as fundamental enablers. Qt uses aforementioned enables to give use higher-level UI and application development components. It includes the core event loop, signals and slot mechanisms. As a cross-platform framework it provides abstraction for Unicode, threads, mapped files, shared memory, regular expressions and many more.
- *QtWidgets* is more of a traditional module with UI elements to create classic desktop-style user interface. It allows user to render to the screen, handle user input events. All UI elements provided by Qt are either subclasses of QtWidgets or use some form of connection with a QWidgets class [27, 28].

Core usage and rendering example can be also seen in the `main` part of WEnux application, showcased in the listing 3.1.

Listing 3.1: Top-level script of WEnux with PyQt5 usage

```
from Gui.Gui import WenuxGUI
from PyQt5 import QtWidgets

        if __name__ == "__main__":
        app = QtWidgets.QApplication([])
        ...
        application = WenuxGUI()
        application.show()
        sys.exit(app.exec())
```

Qt Designer is a tool for designing and building Qt graphical user interfaces with QtWidgets. Once can create and heavily customize windows, dialogs or buttons in a WYSIWYG editor, that serves also as a testing tool. GUI created by Qt Designer is easy to integrate with Python code, using Qt's signals and slot mechanisms.

Result of UI, designed in Qt Designer is an XML `.ui` file format without any additional code. Therefore, Qt includes `uic` utility to generate a C++ code that actually creates the GUI. `QUiLoader` class is the one, that allows an application to load a `.ui` file and to create the corresponding UI dynamically.

However, PyQt5 does not provide a wrapper for `QUiLoader`, but instead incorporates `uic` Python module, which can load `.ui` files as well to dynamically generate user interface.

WANem was mostly designed and tested in Qt Designer and with UI itself was generated by following utility - `pyuic5 qt5.ui -o Qt5.py`. After this step, it was possible to utilize all UI related objects in Python code and connect previously written WANem implementation.

Bridge

In order to simplify emulation to one point of configuration, dedicated interfaces for emulation are bridged together. This process creates a shared interface and requires `bridge-utils` package, containing userspace utility `brctl`.

For this purpose a dedicated shell script has been written, that takes three arguments containing names of mentioned interfaces and intent of the operation, that can be either setup or teardown. Script `bridge.sh` cleans up previous configuration, sets up a new bridge with two given interfaces and assigns it an IP address. After the WEnux finishes, the script is called again for a teardown on created bridge.

Core logic

Depending on the direction of impairment, type of device is selected - either IFB in ingress path, replacing default qdisc, or real device with egress path. To allow higher granularity of configuration prio qdisc is created in order to allow some traffic to

bypass impairments. Next in order are the traffic control filters for inclusion, exclusions, different IP versions and ports. These filters are referring to their parent root qdisc, where `flowid 1:3` is used for inclusion, while `flowid 1:2` serves exclusion of specified filter rules. Impairments themselves follow in the next step, binding new qdisc to their parent root qdisc with all user defined arguments. Available impairments to be emulated are - packet loss, duplication, delay, jitter, reordering, link rate and lastly packet overhead. All of aforementioned impairments can be selectively applied on included IP addresses and ports in both directions of traffic flow.

Listing 3.2: Method netem in the TcWrapper class

```
def netem(self, operation):
        ...
        loss = Loss(self.loss_ratio,
                    self.loss_corr)

        dupl = Duplicate(self.dupl_ratio,
                         self.dupl_corr)

        delay = Delay(self.delay,
                      self.jitter,
                      self.delay_jitter_corr)

        reorder = Reorder(self.reorder_ratio,
                          self.reorder_corr)

        corrupt = Corrupt(self.corrupt_ratio)

        rate = Rate(self.limit,
                    self.overhead)

        netem_collection = NetemCollection(self.nic)
        netem_collection.extend([loss, dupl, delay,
                                 reorder, corrupt, rate])
        netem_collection.apply_cmd(operation=operation)
```

Core logic is contained mostly in the `TcWrapper` module, that does the initialization of prio qdisc and additional ingress related operations if needed. It defines `tbf` and `netem` methods dedicated for the direct control of their tc userspace utility and fabrication of correct command for the given impairment. This class creates an instance for each of the configurable netem parameter as a child dataclass to the `NetemCmd` and hands them over to the `NetemCollection` class, that aggregates all netem parameters and their generated command to master one, which is executed through method `apply_cmd`. As an example in the figure 3.3, `Delay` dataclass is shown with basic functionality to compose the partial command.

Listing 3.3: Delay dataclass that inherits from the NetemCmd

```
@dataclass(eq=True, frozen=True)
class Delay(NetemCmd):
        ...
        _time: int = 0
        _jitter: int = 0
        _correlation: float = 0.0

        def make_cmd(self):
                _time = self.format(self._time, "ms")
                _jitter = self.format(self._jitter, "ms")
                _correlation = self.format(self._correlation, "%")

        return f"delay {_time} {_jitter} {_correlation}"
```

`NetemCollection` is a child class to `TcCollection` and its main use is to allow for application of aggregated command. This is done through keeping the reference to all sub-commands classes created in the `TcWrapper`'s method `netem`. To aggregate all the created commands, the `apply_cmd` method looks for all of its own child dataclasses of `NetemCmd`. If dataclass is not empty, its result, consisting of partial netem command, is taken and appended to the final command to be run. The inner workings can be seen in the listing 3.4.

Listing 3.4: NetemCollection class used for aggregation of NetemCmd classes and execution of netem command

```
class NetemCollection(TcCollection):
        ...
        def apply_cmd(self, operation):
        ...
        _used_dtcls = set()

        self._operation_check(operation)

        for dtcls in self:
                for param_val in dtcls.__dict__.values():
                        if param_val:
                                _used_dtcls.add(dtcls)

        ...

        check_call(f"tc qdisc {operation} dev {self._nic}
        parent 1:3 handle 30: netem " +
        " ".join([x.make_cmd() for x in _used_dtcls]))
```

3.3.2 Installation

WEnux requires few extra modules to function properly, such as its ingress datapath impairment. As application was developed under Fedora 30 and 31, with some

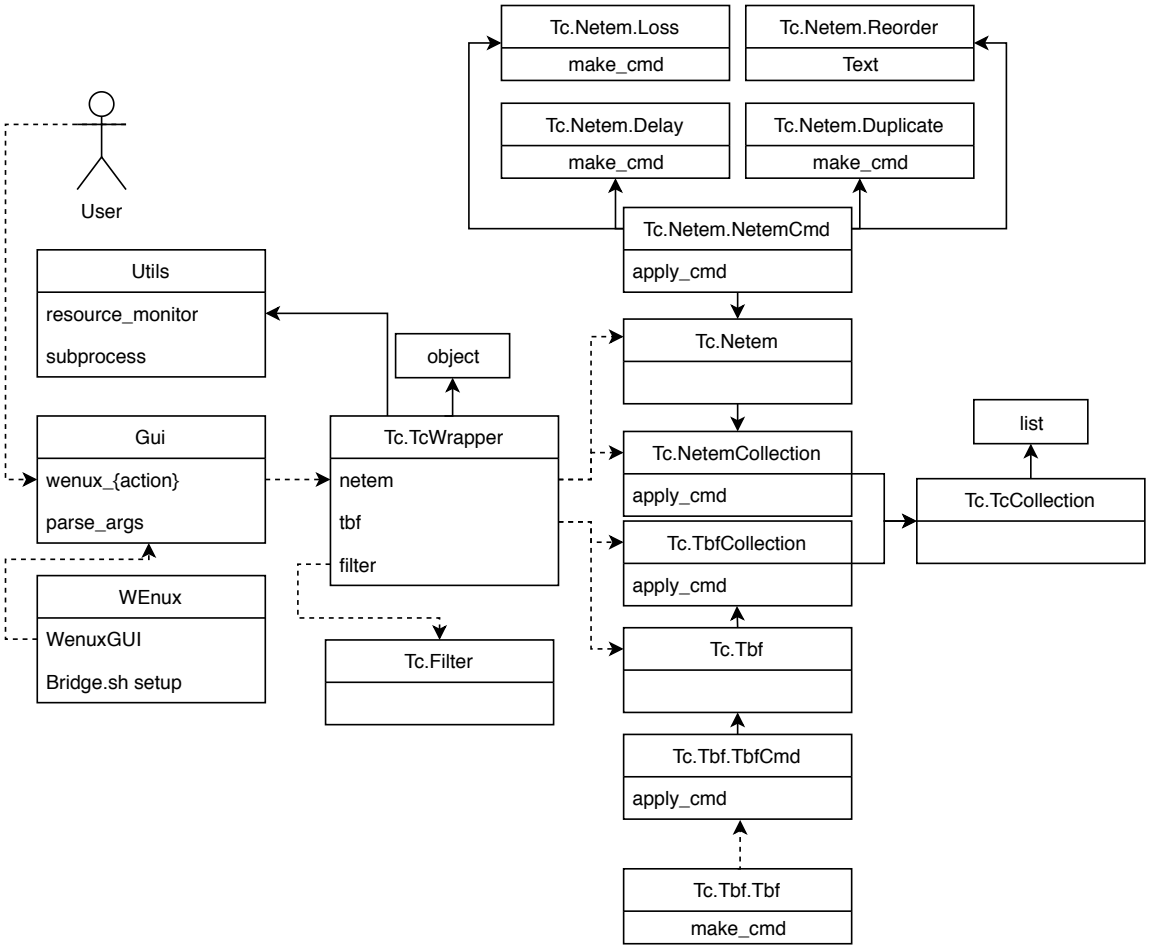

Fig. 3.4: Final specimen design of WEnux

testing done on RHEL8.1 and RHEL8.2, it makes the most sense to explain the installation process with regards to RPM-based Linux distributions.

Due to the WEnux code using *fstrings* and *dataclasses* it is required to run either `Python3.6` with dataclasses installed additionally through *pip* or natively `Python3.7` that has dataclasses included in the built-in packages.

Core package that does the emulation in WEnux is contained in `iproute-tc`, including tc module with netem, tbf, filter and many others. For ingress impairments `ifb` module is required to be loaded into the kernel. However there is a Fedora specific package that user needs to get before being able to successfully manipulate ingress traffic. Depending on the type of installation of RPM-based distribution packages that need to be present are `kernel-modules-extra` and `kernel-debug-modules-extra`. For the bridging capability user needs to install `bridge-utils` package, containing userspace utility `brctl`, utilized in the `bridge.sh` script used in the WEnux initialization.

For the graphical interface of WEnux it is required to have `PyQt5` package, that can be installed either with *pip* under the name of `pyqt5` or directly through system repository through *dnf* utility under the name of `python3-qt5`.

WEnux as a whole can be installed either through *pip*, located in Python package index - *PyPI* repository available online. Or it can cloned from *git* repository. Last option is available to get WEnux is as an appendix in the thesis itself, navigating into `python` folder. WEnux does not have any dependency on other Python libraries and it only uses built-in modules that already come with standard installation of Python.

Lastly WEnux requires root access due to manipulation with kernel.

Installation and running process (some steps might be redundant based on distribution and presence of WEnux package):

1. `dnf install kernel-modules-extra kernel-debug-modules-extra -y`
2. `dnf install python3.7 -y`
3. `dnf install iproute-tc -y`
4. `dnf install bridge-utils -y`
5. `dnf install python3-qt5 -y`
6. `mkdir wenux; cd wenux/; pip3 install -t .; chmod u+x wenux.py`
7. `./wenux.py {interface1} {interface2}`

4 Testing

This chapter concludes a small form of validation phase that has been done to the WEnux. Different parts of emulation are briefly tested with sample CPU and memory usage.

First off, there is a need to explain expected outcome of the network emulator. Selection of the language does not play any role here with current implementation, because it does not control the packet flow or quality of service parameters. Traffic control is done directly through kernel with heavily optimized software implementation, which speed would not be surpassed without using some form of hardware acceleration or for example defining XDP. Also, software emulation is extremely costly procedure when it comes to high speed networking and one should not expect that it will replace a hardware emulator, with dedicated and optimized hardware setup with acceleration of different parts of emulation.

As WEnux is directly using kernel modules to control ingress and egress traffic, the implementation and emulation itself is extremely accurate up to a point, where kernel has enough resources to manipulate the traffic, such as storing it to cause delay, jitter, reorder and other types of impairments.

Due to the use of IFB in the ingress flow scenario, both flows are similar in terms of resource usage. In this case, redirecting traffic from an interface's ingress to the IFB's egress is done directly by stealing the packet and placing it straight to the egress of the IFB device. Moreover, this operation is not that costly in term of resources.

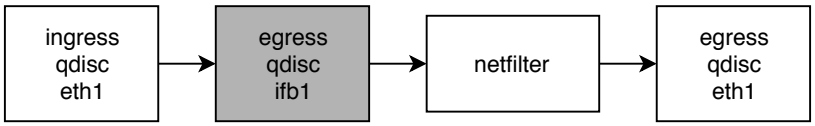

Fig. 4.1: Packet's traversal through kernel with the use of IFB

Even though WEnux is directly plugging into kernel, emulation itself introduces some deviation, caused by processing and execution of additional code introduced by either `netem` or other extensions of `tc`. One thing to note is that some parameters, for example rate or limit throttling, delay and others are affected by several factors, such as kernel clock granularity, which causes imperfections in shaping in some scenarios. This can be seen in an bursts of packets, known as artificial packet compression. Another limitation, which introduces artificial delay, is in the network adapter buffers.

4.1 Testing methodology

WEnux test were conducted on a *ThunderX2 CN9980 - Cavium* [29] ARM server, using microprocessor based on Vulcan microarchitecture. This is well known high-performance ARM server, manufactured by *TSMC*, using standard `64 bit` word size, with 32 cores and 128 threads per socket. This server was chosen due to being available at the time for testing, and to also prove, that WEnux is going to function under *kernel-alt*. Used system was RHEL8 General Availability release, with default kernel options and parameters.

However WEnux was developed under the *kernel-x86-64* and locally tested on Fedora 30 and 31 distributions.

Implementation of emulator was tested with artificially generated UDP[1] traffic with packet size of 1000 bytes at the set packet rate of 300 packets per second, with in-between packet delay of 3.33ms, producing a total traffic flow of 300 kilobytes per second. This stream was generated with iperf3[2] framework.

Based on recommendations from the RFC 2679 [30] one-way delay was defined as

$$Delay = T_r - T_s$$

where T_s represents the time, when source sends the first bit of the packet, and T_r is the time, when last bit of the packet is received by the destination.

Definition of one-way delay jitter is derived from the delay definition itself, measured as the absolute value of the difference between the delay of two consecutive packets.

$$Jitter = |D_n - D_{n-1}|$$

formula is straight-forward, where D_n describes the one-way delay of the n-th packet and D_{n-1} is the exact same delay of the *(n-1)*-th packet respectively.

Packet loss is calculated as a portion of lost packets for a sliding window of one second in length.

Standard deviation (σ) of delay, implemented as *correlation* parameter in *WEnux*, is calculated based on the formula of the one-way delay as

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(D_i - \bar{D})^2}$$

where N is the number of delay samples used in σ calculation, D_i represents delay of the i-th sample of delay, and lastly the $\bar{D}$ is the mean value of the one-way delay samples.

[1] UDP stream was used due to TCP having congestion control algorithms, which would interfere with the results

[2] `https://iperf.fr/`

WEnux was tested for delay generation accuracy by setting a reference value in emulator and later the value of one-way delay was measured for comparison. Tests were run for 30 seconds, sample rate was set to 1/10 of a second for total of 300 samples. It might seem that sample rate, as well as, test duration are insufficient to evaluate the emulator, but for the sake of this work it should be enough to estimate the behaviour of the implementation. To stress the buffers and kernel itself, test was run with reconfiguration of WEnux after 5 seconds, to double the parameter value of previous iteration, with seed value set to 1 ms.

4.2 Evaluation

Recorded and reference values for each of measured parameters are shown in figures 4.2, 4.3 and 4.4. All of these graphs are showing mostly extreme values, where kernel has issues handling almost real-time emulation with minimal impairments, with high level of error. As parameters rose, accuracy of emulation was getting higher as well, with error rate dropping below 1% at around 50 ms of delay.

This study presents an extreme cases in terms of low parameter values, which most probably wont be used, as they are not real scenarios in WAN. However, going above the 50 ms is presenting such stable results, that it does not make sense to evaluate the accuracy of emulation.

On the other end of spectrum, using values above 1000 ms, can stress resources available, but considering the server used for test, the change was marginal, or even negligible.

Moreover, to evaluate the CPU and RAM consumed by the WEnux, there is need to change the whole methodology, to push the maximum throughput available through the NIC and gradually reduce the rate to find the threshold of rate that the server is able to handle with WEnux. Anyhow, rate is only one parameter, creating one combination of a large scale, that WEnux currently offers. In depth evaluation of resource usage is beyond the scope of this work.

However, some combinations were tested and their respective values were recorded in a table 4.1. Chosen methodology was very simple, consisting of recording the baseline CPU and RAM usage before using the WEnux and running iperf3 UDP stream limited to 100Mbps. Then, the same test was run with WEnux enabled with different combination of parameters and unlimited UDP stream.

Difference between baseline value and WEnux test value is recorded in the table for respective parameters.

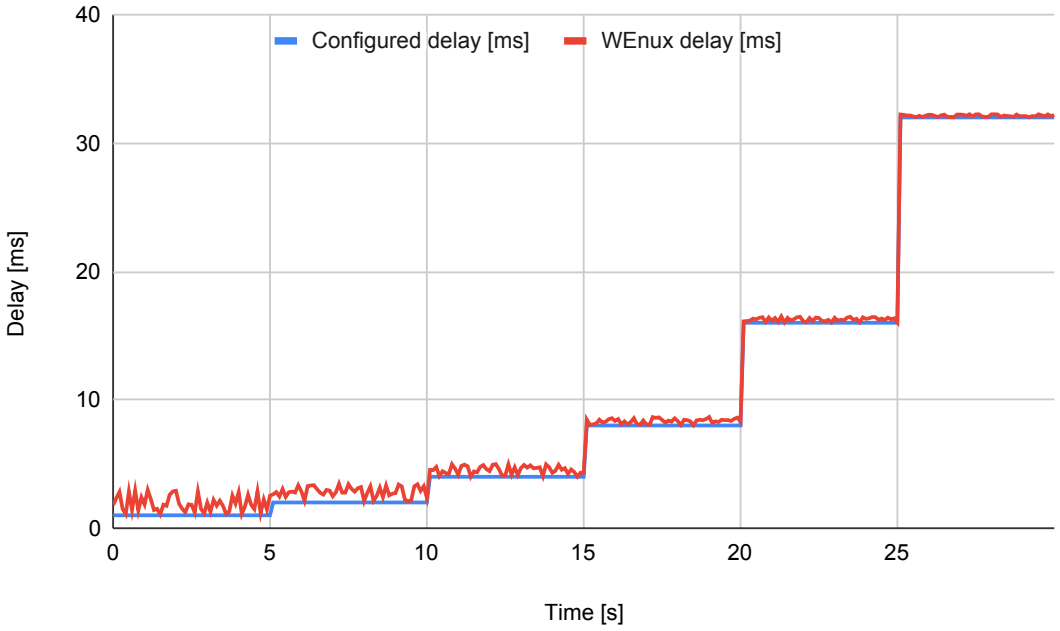

Fig. 4.2: Measured and configured delay

Tab. 4.1: CPU and RAM usage of WEnux emulations

Combination of WEnux parameters	**CPU [%]**	**RAM [GB]**
`limit` 100000, `delay` 200, `jitter` 20	1.3%	0.02
`limit` 100000, `delay` 500, `jitter` 20	1.5%	0.06
`limit` 100000, `delay` 1000, `jitter` 20	1.7%	0.11
`limit` 100000, `loss_ratio` 1	0.3%	0.00
`limit` 100000, `reorder_ratio` 1	2.0%	0.02

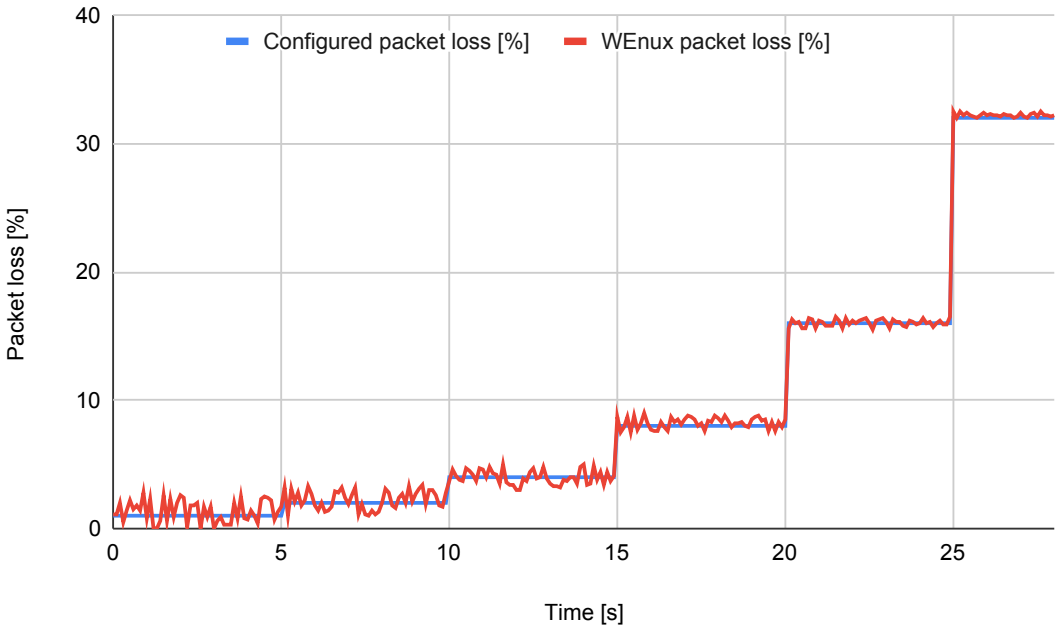

Fig. 4.3: Measured and configured packet loss

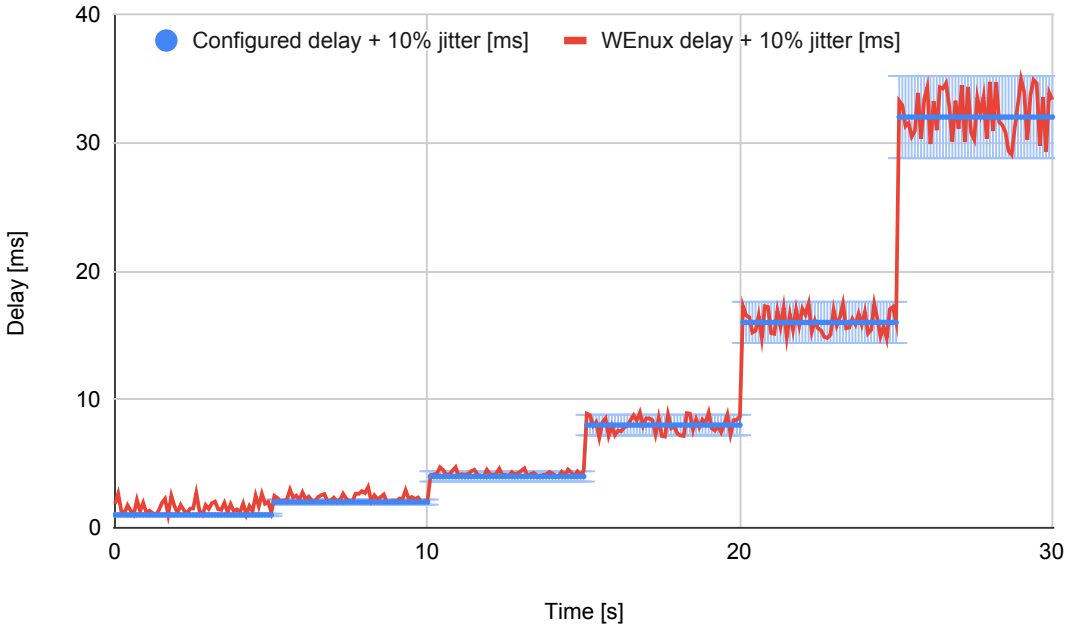

Fig. 4.4: Measured and configured delay with 10% jitter

Conclusion

This thesis elaborates on kernel networking along with virtualization in networks. Kernel network is the core focus of the work, as its knowledge was required for the development of emulator. First chapter discusses Linux network stack, explaining the ISO/OSI network model, needed for the correct definition of parameters of the emulator. Kernel modules utilized in the emulation are explained in depth, especially *Netfilter*, which is the essential part of this work. Existence of Netfilter's hooks, present in between the transport layer of ISO/OSI model and NIC, is what allows manipulation with packets close to their raw format on lower layers of the network stack. When packet comes to appropriate Netfilter hook, it is redirected to the traffic control module, that handles it further. Traffic control module, with its internals is examined, as every part of its functionality is used in the final application. In egress data path after packet handover from Netfilter, *classifier*, *scheduler* and *shaper* from traffic control module interact with packet. In ingress path only *policier* is applied, but due to the use of IFB, known as Intermediate Functional Block, even packets coming from ingress path can be treated the same as egress packets. Utilities used from traffic control module are `tc`, `tc-netem` and `tc-tbf`. All of these userspace utilities interact directly with appropriate kernel modules to modify the quality of service parameters of network traffic.

Second chapter discusses the virtualization of network in general, as it gives an overview for models, simulators and emulators. This chapter's purpose is to explain the difference between the simulator and emulator, as well as, understand, why virtualization has limitations and where are its issues.

The main goal of Master's thesis was to create a network emulator able that allows granular control over the delay, jitter, loss and others. The application name WEnux stands for WAN emulator for Linux, and was developed under GPLv3 license. It was developed in Python3.7, initially with command line interface. In the later stages of the work, interface was changed to graphical with the use of Qt. Qt was picked due to being one of the most complete and supported graphical user interface, with good documentation. It offers native design to desktop environment that is being used, whether it is GNOME, KDE or any other Linux desktop environment. Graphical user interface was developed in Qt Designer and made functional with the use of PyQt5 framework. The application itself bridges tested interfaces together to create a single point where the impairments can be applied, but it is also able to function on an individual interface, if needed. Root of the `tc qdisc` tree lies in prio qdisc, on which are filters and other qdiscs bound, creating a hierarchy that allows user to either include some networks or ports to the emulation or to exclude them altogether. Interface of application can be seen in figure 3.2, allowing

configuration of many parameters with interactive feedback from the application in form of the text window, that displays the current status of emulation.

The application was tested according to the fourth chapter, where accuracy of implementation and resource usage were evaluated. These tests were not a formal validation, but are clear indicator, that application is able to emulate the specified WAN very closely, where its accuracy is based on multiple parameters, such as kernel clock granularity, specification of the system where WEnux is running and values of aforementioned parameters. With basic methodology explained in chapter 4.1, it can be said, that application is reasonably accurate and resource efficient.

Bibliography

[1] Beshay, J. D.; Francini, A.; Prakash, R.: On the Fidelity of Single-Machine Network Emulation in Linux. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Oct 2015. ISSN 1526-7539. pp. 19–22. doi:10.1109/MASCOTS.2015.18.

[2] Rosen, R.: *Linux Kernel Networking: Implementation and Theory*. Berkely, CA, USA: Apress. first edition. 2013. ISBN 9781430261964.

[3] Department of information technology, Uppsala university: Exception and interrupt handling. [Online; visited on 29.11.2019]. Retrieved from: `http://www.it.uu.se/education/course/homepage/os/vt18/module-1/exception-and-interrupt-handling/`

[4] Corbet, J.; Rubini, A.; Kroah-Hartman, G.: *Linux Device Drivers, Third Edition*. O'Reilly Media, Inc.. third edition. 2005. ISBN 978-0-596-00590-0.

[5] Price, M.: Navigating the Linux kernel network stack: receive path. [Online; visited on 29.11.2019]. Retrieved from: `https://epickrram.blogspot.com/2016/05/navigating-linux-kernel-network-stack.html?m=0`

[6] *tc(8) - Linux manual page*. Dec 2001. Retrieved from: `http://man7.org/linux/man-pages/man8/tc.8.html`

[7] Gautham, A. V.: The tc command approach. [Online; visited on 29.11.2019]. Retrieved from: `https://github.com/ComputerNetworks-Project/DCE-AQM-Interface/wiki/The-tc-command-approach`

[8] GoldenOak: Linux Kernel Communication - Netfilter Hooks. [Online; visited on 29.11.2019]. Retrieved from: `https://medium.com/@GoldenOak/linux-kernel-communication-part-1-netfilter-hooks-15c07a5a5c4e`

[9] *iptables(8) - Linux manual page*. May 2019. Retrieved from: `http://man7.org/linux/man-pages/man8/iptables.8.html`

[10] Torvalds, L.: Linux. https://github.com/torvalds/linux. 2019.

[11] nftables project. [Online; visited on 27.11.2019].
Retrieved from: https://netfilter.org/projects/nftables/

[12] Gregg, B.: *BPF Performance Tools*. Addison-Wesley Professional. first edition. 2019. ISBN 978-0-13-655482-0.

[13] Bernat, V.: IPv4 route lookup on Linux. [Online; visted on 4.12.2019].
Retrieved from: https://vincent.bernat.ch/en/blog/2017-ipv4-route-lookup-linux#implementation-in-linux

[14] Bernat, V.: IPv4 route lookup on Linux. [Online; visited on 29.11.2019].
Retrieved from:
https://vincent.bernat.ch/en/blog/2017-ipv4-route-lookup-linux

[15] Brown, M. A.: Guide to IP Layer Network Administration with Linux.
[Online; visited on 5.12.2019].
Retrieved from: http://linux-ip.net/html/index.html

[16] Brown, M. A.: Traffic Control HOWTO. [Online; visited on 2.12.2019].
Retrieved from:
http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html

[17] Boismenu, F.: Demystification of TC. [Online; visited on 1.12.2019].
Retrieved from: https://medium.com/criteo-labs/demystification-of-tc-de3dfe4067c2

[18] Amante, S.; Carpenter, B.; Jiang, S.; et al.: IPv6 Flow Label Specification. 6437. RFC Editor. November 2011. ISSN 2070-1721.

[19] Mikrotik: Manual:HTB-Token Bucket Algorithm. [Online; visited on 3.12.2019].
Retrieved from:
https://wiki.mikrotik.com/wiki/Manual:HTB-Token_Bucket_Algorithm

[20] The Linux Foundation: ifb. [Online; visited on 4.12.2019].
Retrieved from: https://wiki.linuxfoundation.org/networking/ifb

[21] Claeskens, G.; Hjort, N. L.: *Model Selection and Model Averaging*. Cambridge University Press. 2008. ISBN 9780521852258.

[22] Rosbach, M.: *Verification of Network Simulators*. Master's Thesis. University of Oslo. Norway. 2012.

[23] Koninklijke Bibliotheek: What is emulation? [Online; visited on 27.11.2019]. Retrieved from: https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation

[24] Rowland, T.; Weisstein, E. W.: Church-Turing Thesis. [Online; visited on 27.11.2019].
Retrieved from:
http://mathworld.wolfram.com/Church-TuringThesis.html

[25] *tc-netem(8) - Linux manual page*. November 2011.
Retrieved from:
http://man7.org/linux/man-pages/man8/tc-netem.8.html

[26] Kalitay, H. K.; Nambiarz, M. K.: Designing WANem : A Wide Area Network emulator tool. In *2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011)*. Jan 2011. ISSN 2155-2509. pp. 1–4. doi:10.1109/COMSNETS.2011.5716495.

[27] Riverbank Computing: PyQt v5.14.1 Reference Guide. [Online; visited on 25.5.2020].
Retrieved from:
https://www.riverbankcomputing.com/static/Docs/PyQt5/

[28] The Qt Company: Qt Documentation. [Online; visited on 25.5.2020].
Retrieved from: https://doc.qt.io/

[29] WikiChip: ThunderX2 CN9980 - Cavium. [Online; visited on 10.12.2019].
Retrieved from:
https://en.wikichip.org/wiki/cavium/thunderx2/cn9980

[30] Almes, G.; Kalidindi, S.; Zekauskas, M.; et al.: A One-Way Delay Metric for IP Performance Metrics (IPPM). 81. RFC Editor. January 2016. ISSN 2070-1721.

List of symbols, physical constants and abbreviations

API	Application Programming Interface
BPF	Berkeley Packet Filter
CIDR	Classless Inter-Domain Routing
DoS	Denial of Service
DS	Differentiated services
DES	Discrete Event Simulator
FIB	Forwarding Information Base
FIFO	First in, first out
GUI	Graphical User Interface
HTB	Hierarchy Token Bucket
HTTP	Hypertext Transfer Protocol
IFB	Intermediate Functional Block
ISO	International Organization for Standardization
LAN	Local area network
NIC	Network Interface Card
OSI	Open Systems Interconnection
PBR	Policy-based routing
RPDB	Routing policy database
SKB	Socket buffer
SOHO	Small office, home office
ToS	Type of Service
TC	Traffic Control
TBF	Token Bucket Filter
UDP	User Datagram Protocol
qdisc	Queueing discipline
WAN	Wide area network
WYSIWYG	What you see is what you get
XML	eXtensible Markup Language

List of appendices

A CD content

Included CD contains source codes of this thesis and source codes of the WEnux application. The structure of important files on the CD is shown below. Several files are omitted to reduce size and complexity of the directory tree.

```
/
├── images/ ..... Images and diagrams used by latex
├── latex/ ..... All latex related files to generate the thesis
│   └── text/ ..... Text of the thesis
├── pdf/ ..... Final generated pdfs
└── python/ ..... source codes of WEnux
    ├── bridge.sh ..... Shell script used for managing the bridge device
    ├── Gui/ ..... Control and connection of WEnux, Qt related code
    │   ├── Gui.py ..... Core part of logic connecting the Gui with functionality
    │   ├── Qt5.py ..... uic Python file generated from pyuic5
    │   └── qt5.ui ..... XML file created by Qt Designer
    ├── README.md
    ├── setup.py
    ├── Tc/ ..... WEnux main functionality related to tc modules
    │   ├── Filter.py
    │   ├── Netem.py
    │   ├── Tbf.py
    │   ├── TcCollection.py
    │   └── TcWrapper.py
    ├── Utils/ ..... Basic Python utilities used by WEnux
    └── wenux.py/ ..... Entrypoint of WEnux, initialization and main
```